W4995 Applied Machine Learning

Neural Networks

04/12/17

Andreas Müller

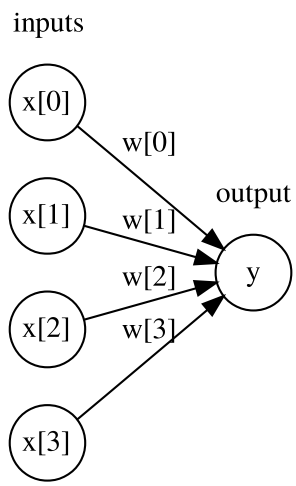
History

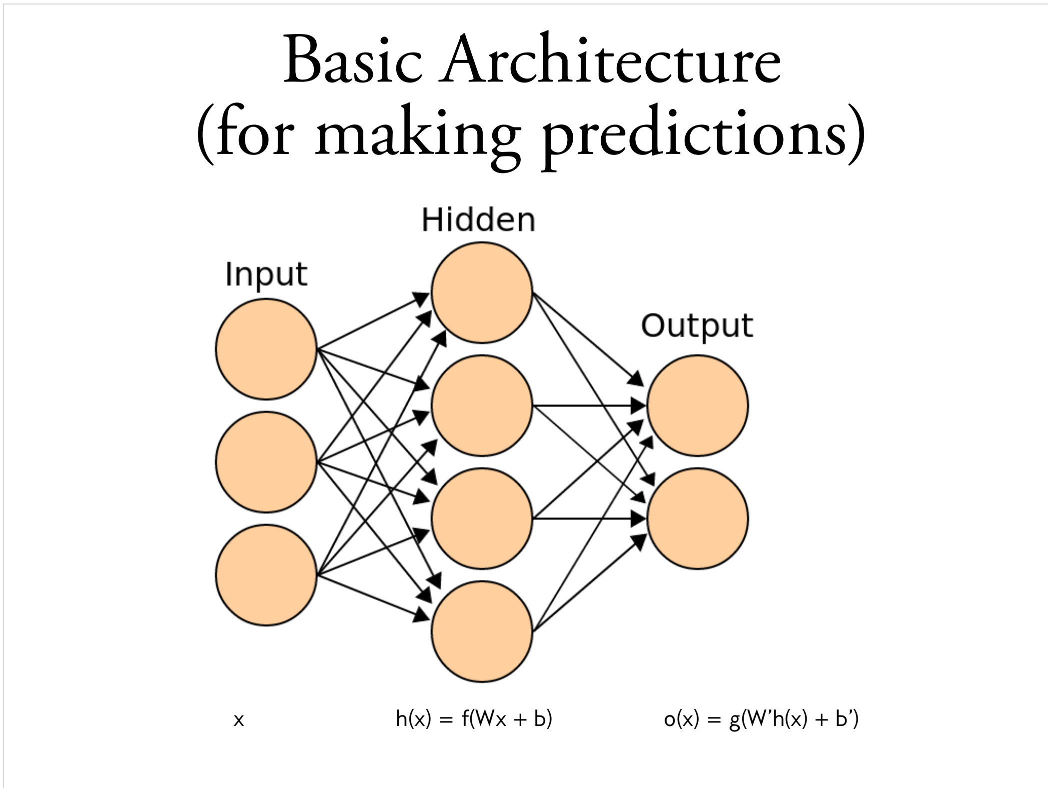
- Nearly everything we talk about today existed ~1990
- What changed?
  - More data
  - Faster computers (GPUs)
  - Some improvements:
    - relu
    - Drop-out
    - adam
    - batch-normalization

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, general adverserial networks, neural turing machines...)

Logistic regression drawn as neural net





First let's describe how to make a prediction given a model.

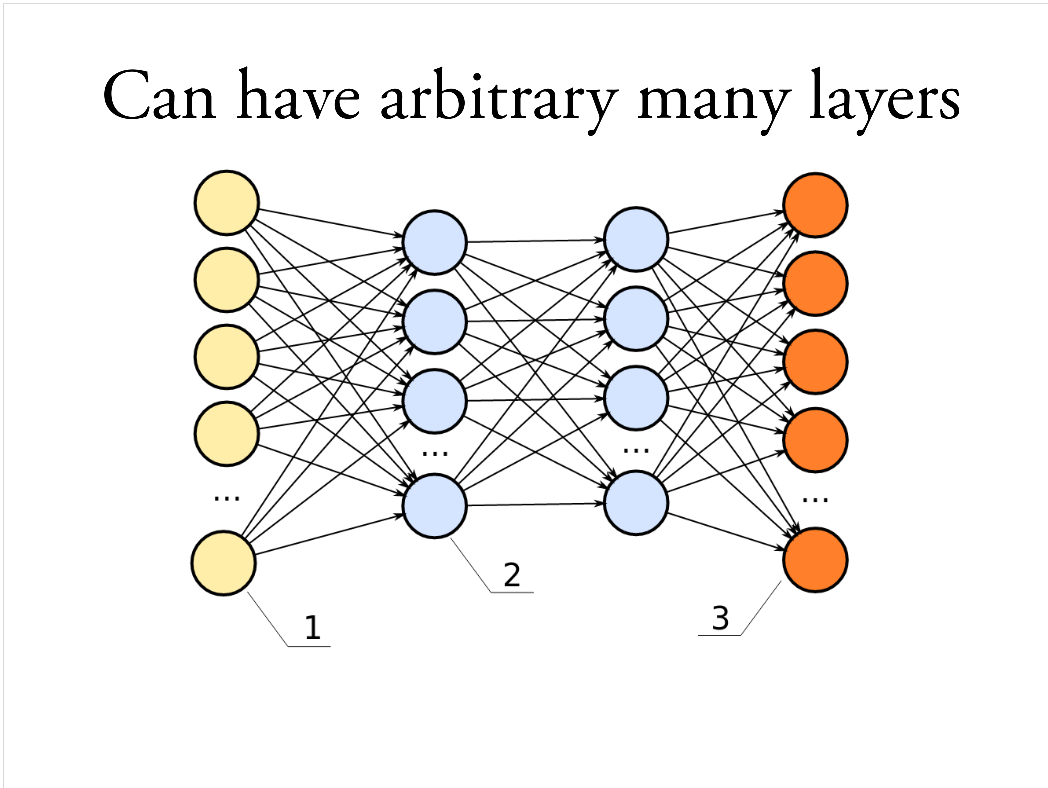
Input denotes single sample, here three input features.

Hidden layer here 4 units is matrix multiply with W, b added (size of b is 4 here), followed by the univariate non-linear function f – sigmoid, tanh, rectifying linear function.

Output is a matrix multiplication with different weight matrix W', b' added (size 2 here), followed by another non-linear function g. The function g for the output layer is often different: identity for regression, soft-max for classification.

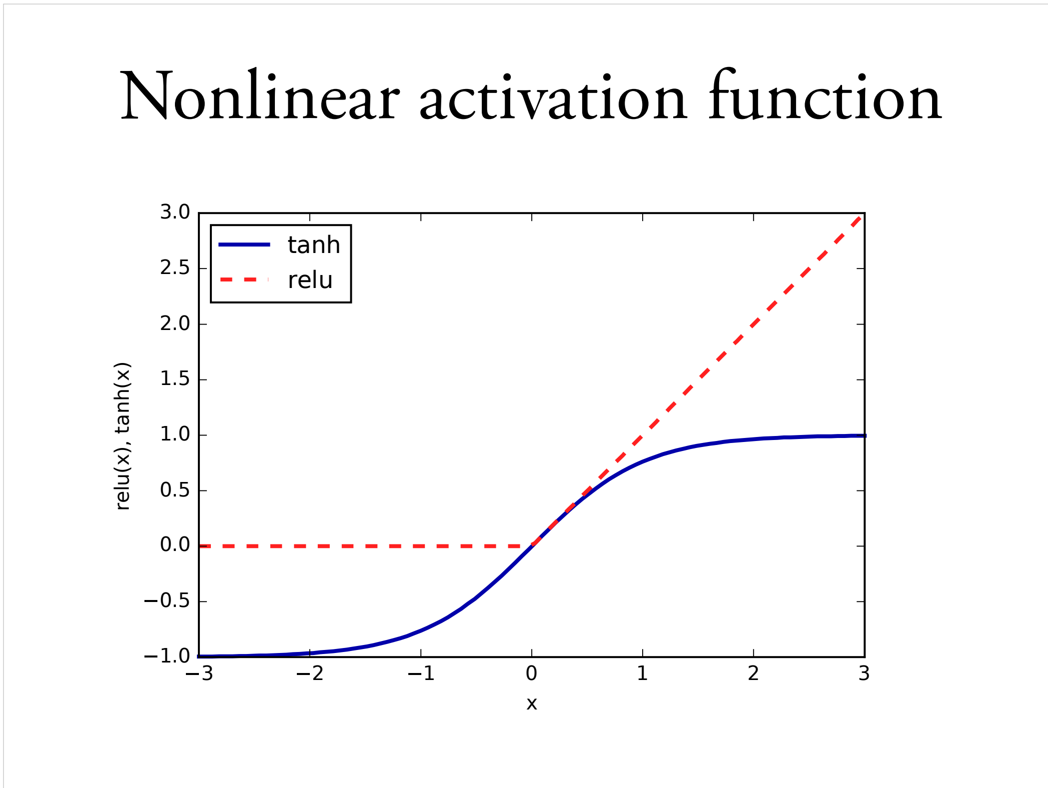
We want to learn W (3x4) W' (4x2), b (4,), b' (2,).

Can think of it as logistic regression with learning a non-linear basis transformation.



Hidden layers usually all have the same non-linear function, weights are different for each layer.
Many layers → “deep learning”.
This is called a multilayer perceptron, feed-forward neural network, vanilla feed-forward neural network.

For regression usually single output neuron with linear activation.
For classification one-hot-encoding of classes, n_classes many output variables with softmax.



Choices for activation function f of hidden layers.
Traditional tanh (or logistic sigmoid, not shown, but similar).
Tanh squashes to open interval (-1, 1).
Relu – recent trend, linear function x=y for positive, constant for negative values. Bias allows shifting the cut-off (~ linear splines)

Training objective

$$h(\mathbf{x}) = f(W_1\mathbf{x} + \mathbf{b}_1)$$

$$o(\mathbf{x}) = g(W_2 h(\mathbf{x}) + \mathbf{b}_2) = g(W_2 f(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$$

$$\min_{W_1, W_2, \mathbf{b}_1, \mathbf{b}_2} \sum_{i=1}^{N} \ell(y_i, o(\mathbf{x}_i))$$

Could add regularization

$$= \min_{W_1, W_2, \mathbf{b}_1, \mathbf{b}_2} \sum_{i=1}^{N} \ell(y_i, g(W_2 f(W_1\mathbf{x}_i + \mathbf{b}_1) + \mathbf{b}_2))$$

ℓ squared loss for regression
cross-entropy loss (multi-class log-loss) for classification

Backpropagation

- For gradient based-method need $\frac{\partial o(\mathbf{x})}{\partial W_i}$ $\frac{\partial o(\mathbf{x})}{\partial \mathbf{b}_i}$

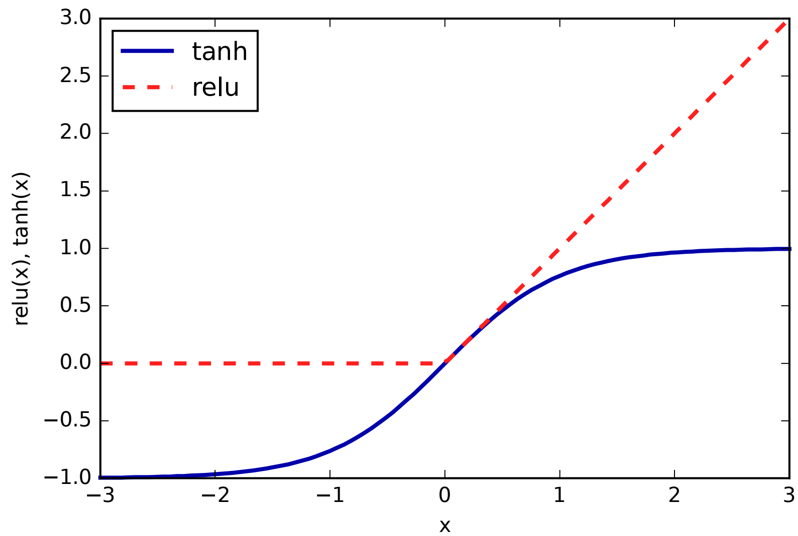
$$\mathrm{net}(\mathbf{x}) := W_1\mathbf{x} + b_1$$

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

Backpropagation = Chain Rule + Dynamic Programming

- Easy to write down for last layer
- Example for squared loss (g is identity for regression)
- Can use the chain rule to compute other gradients
- Bottom layers require partial derivatives of upper layers → reuse results
  - Backpropagation:
    dynamic programming + chain rule
  - “backward pass” compute partial derivatives starting at the last layer.
  - You should try to go through that yourself once

But wait!



- relu is not differentiable.
- But it’s differentiable almost anywhere.
- “subgradient descent” - little issues in practice

Optimizing W, b

$$W_i \leftarrow W_i - \eta \sum_{j=1}^{n} \frac{\ell(\mathbf{x}_j, y_j)}{W_i}$$ batch

$$W_i \leftarrow W_i - \eta \sum_{j=k}^{k+m} \frac{\ell(\mathbf{x}_j, y_j)}{W_i}$$ minibatch

$$W_i \leftarrow W_i - \eta \frac{\ell(\mathbf{x}_j, y_j)}{W_i}$$ Online / stochastic

- Standard solvers: l-bfgs, newton, cg
- Problem: Hessian too expensive, can to l-bfgs
- Computing gradients over whole dataset expensive
- Stochastic Gradient Descent to rescue
- Actually use mini-batches

Learning Heuristics

- Constant η not good
- Can decrease η
- Better: adaptive η for each entry if Wi
- State-of-the-art: adam (with magic numbers)

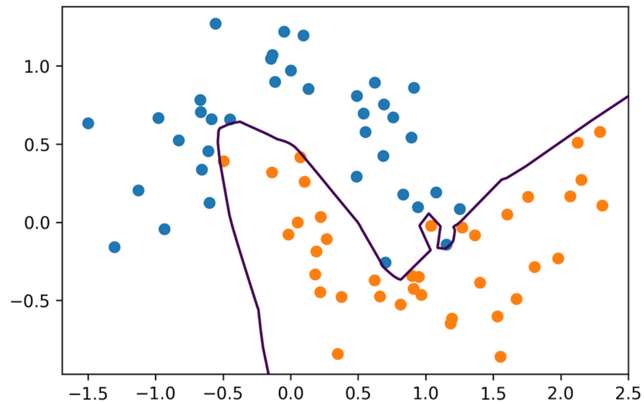
https://arxiv.org/pdf/1412.6980.pdf

http://sebastianruder.com/optimizing-gradient-descent/

Picking optimization algorithms

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

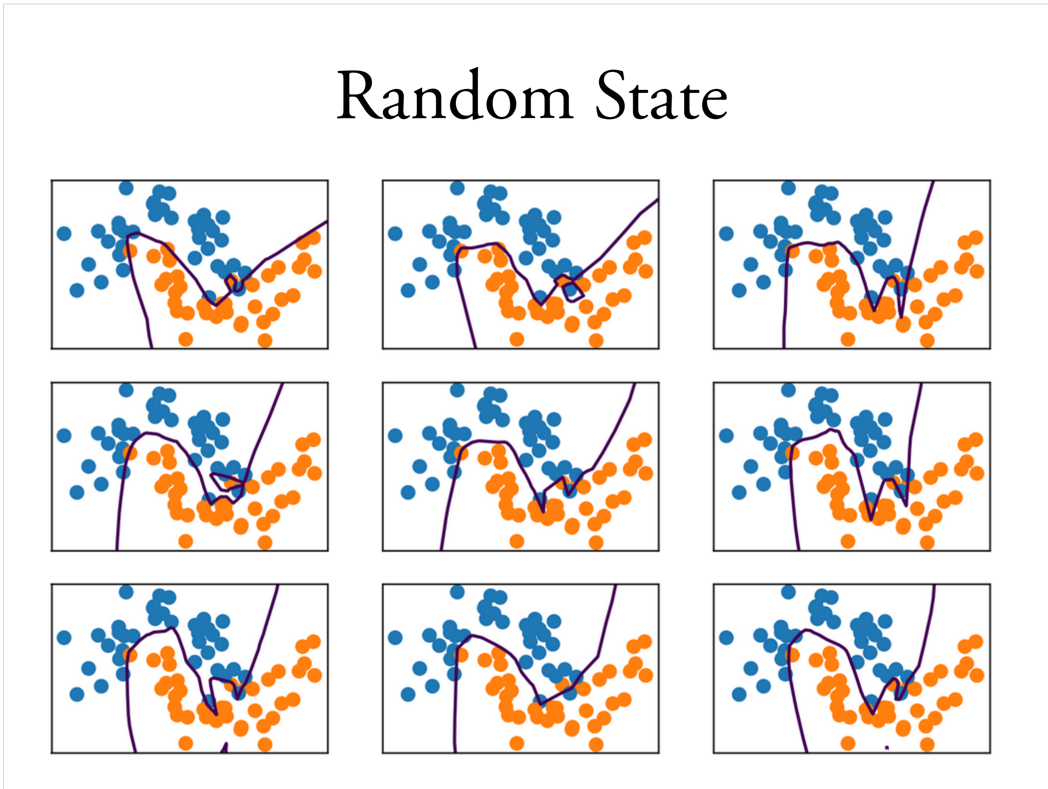
Neural Nets with sklearn




```
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
```

Don’t user sklearn for anything but toy problems in neural nets.



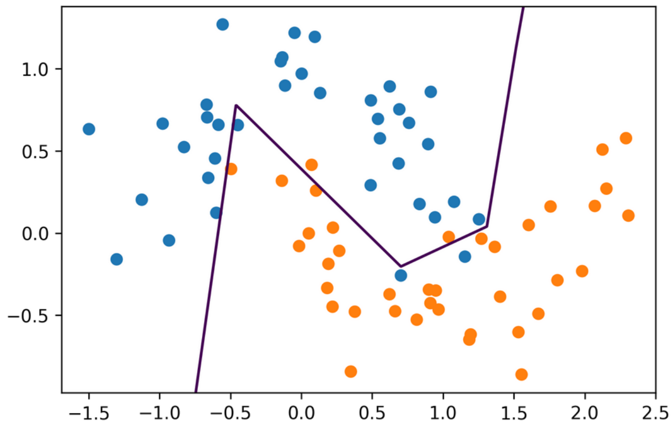
This net is also way over capacity and can overfit in many ways.

Regularization might make it less dependent on initialization.

Hidden Layer Size

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

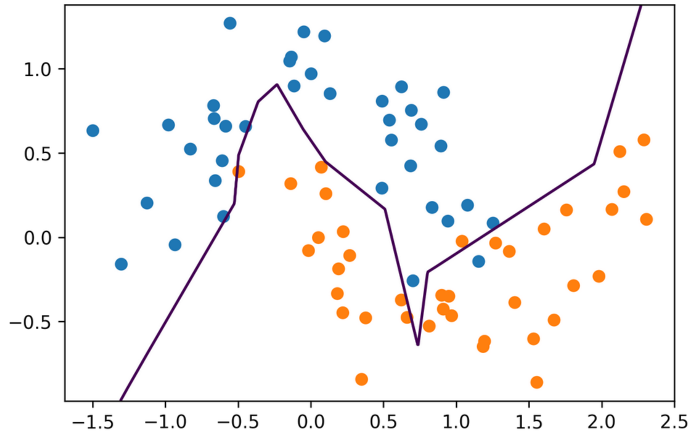
```
0.933333333333
0.92
```



Hidden Layer Size

```
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))
```

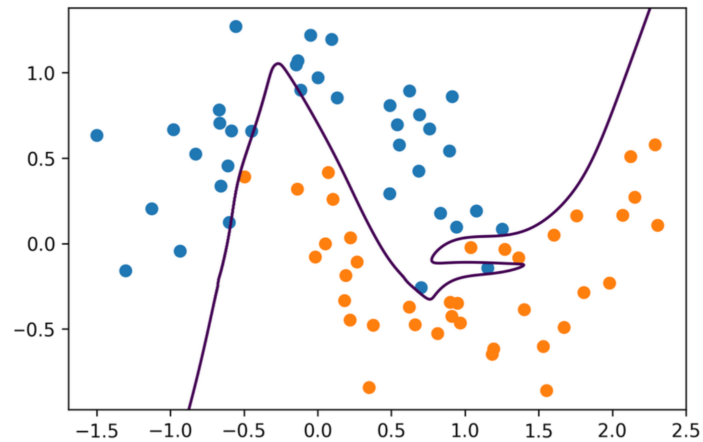
```
0.973333333333
0.84
```



Activation Functions

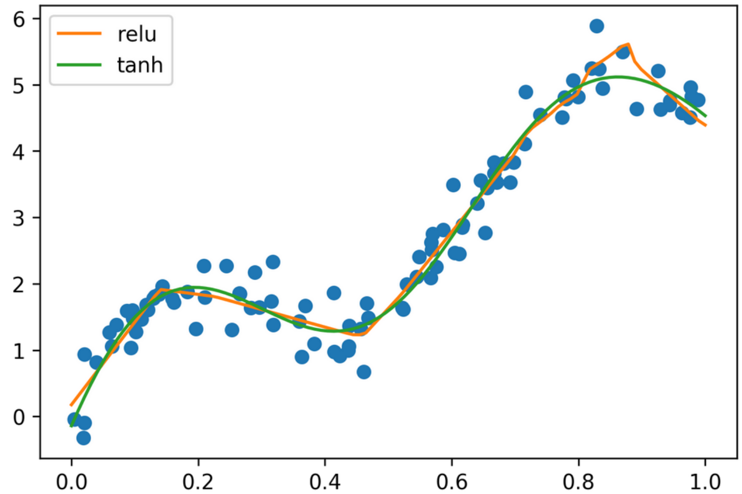
```
mlp = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(10, 10, 10), activation="tanh", random_state=0)
mlp.fit(X_train, y_train)
print(mlp.score(X_train, y_train))
print(mlp.score(X_test, y_test))
```

```
1.0
0.92
```



Using tanh we get smoother boundaries
Here actually it fits the data better.
It might be that relu doesn’t work that well with l-bfgs or with using these very small hidden layer sizes.
For large networks, relu is definitely preferred.

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()
```

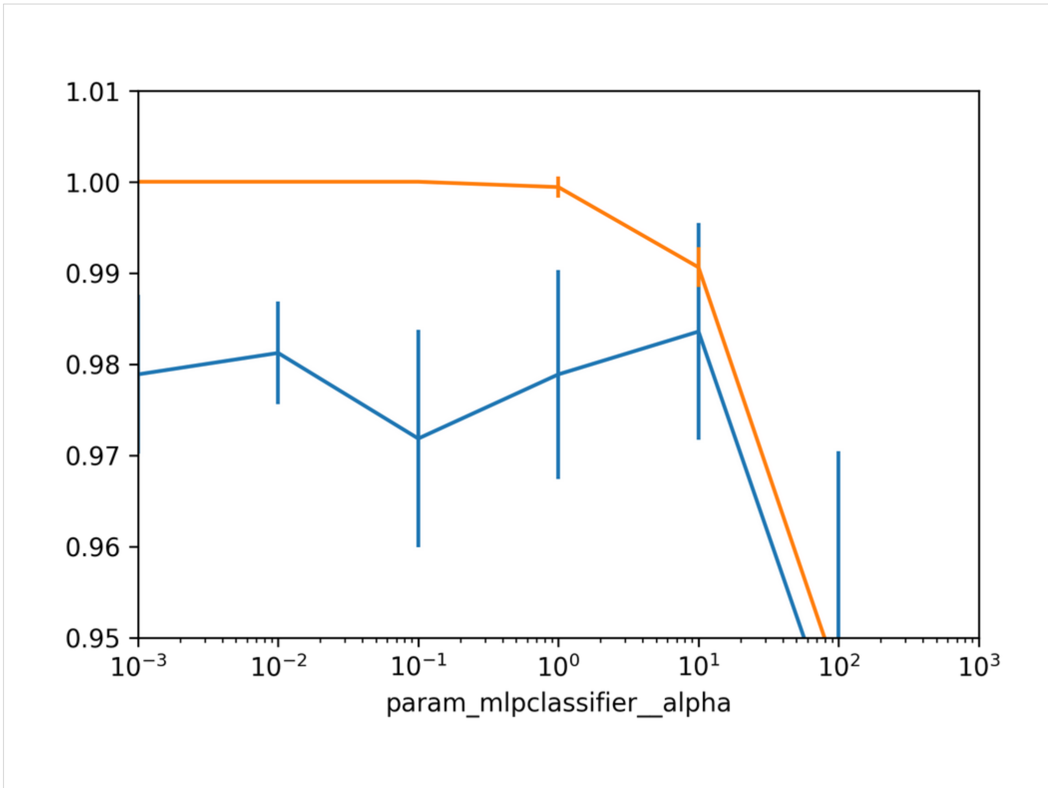
```
X_train, X_test, y_train, y_test = train_test_split(
    data.data, data.target, stratify=data.target, random_state=0)
scaler = StandardScaler().fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
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, cv=5)
```

```
results = pd.DataFrame(grid.cv_results_)
res = results.pivot_table(index="param_mlpclassifier__alpha",
                          values=["mean_test_score", "mean_train_score"])
```

```
res
```

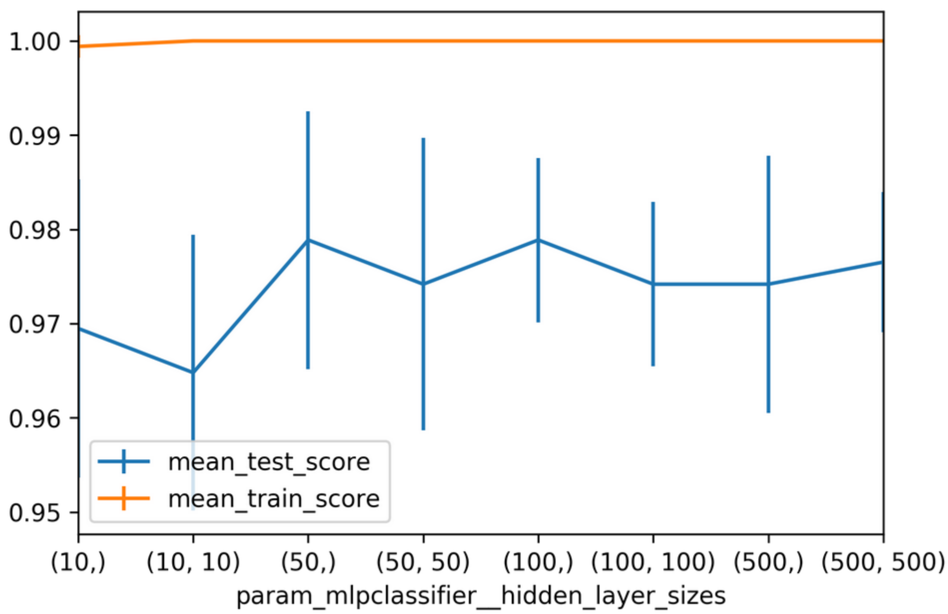
	mean_test_score	mean_train_score
param_mlpclassifier__alpha		
0.001	0.978873	1.000000
0.010	0.981221	1.000000
0.100	0.971831	1.000000
1.000	0.978873	0.999412
10.000	0.983568	0.990612


1.01
1.00
0.99
0.98
0.97
0.96
0.95
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
param_mlpclassifier__alpha

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, cv=5)
```

```
grid.fit(X_train, y_train)
```


1.00
0.99
0.98
0.97
0.96
0.95
mean_test_score
mean_train_score
(10,)
(10, 10)
(50,)
(50, 50)
(100,)
(100, 100)
(500,)
(500, 500)
param_mlpclassifier__hidden_layer_sizes

Getting flexible & 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

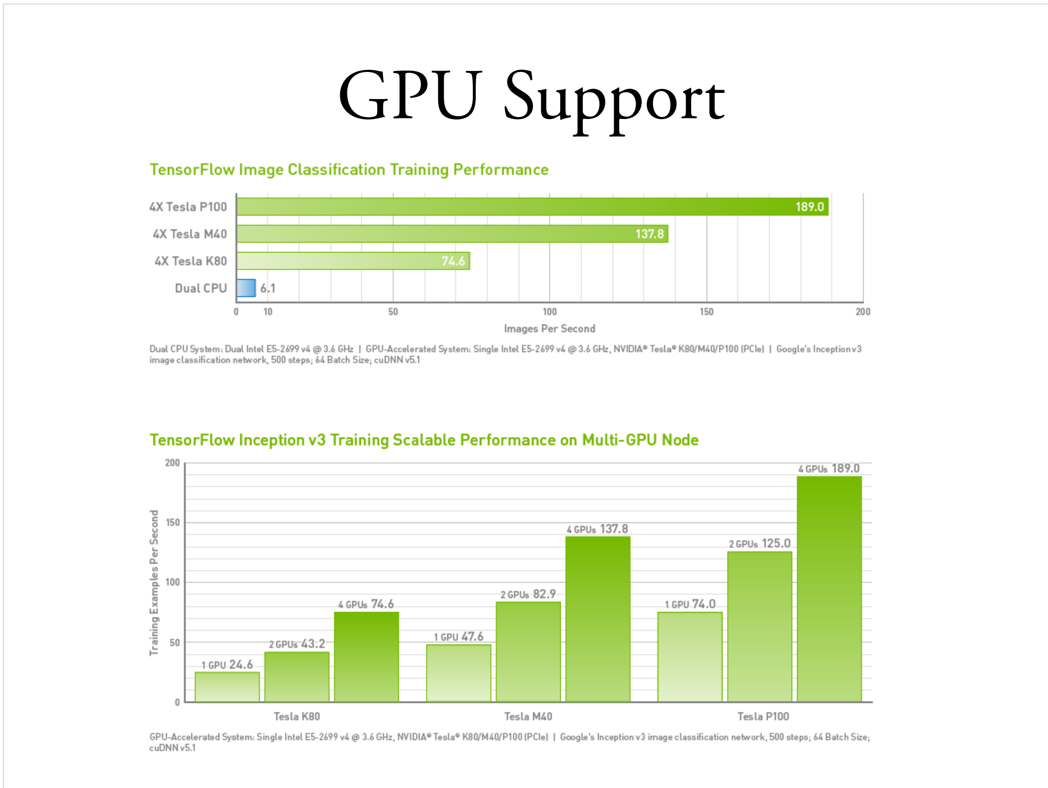
```python
# http://mxnet.io/architecture/program_model.html
class array(object) :
    """Simple Array object that support autodiff."""
    def __init__(self, value, name=None):
        self.value = value
        if name:
            self.grad = lambda g : {name : g}

    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))
            return x
        ret.grad = grad
        return ret
```

```python
# some examples
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))
```

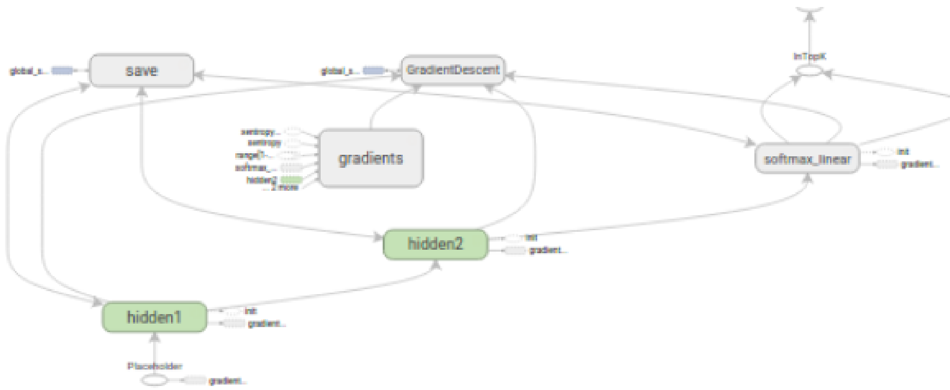
```
[4 9]
{'b': array([1, 2]), 'a': array([3, 4])}
```



From
http://www.nvidia.com/object/gpu-accelerated-applications-tensorflow-benchmarks.html

Take with a grain of salt.

Computation Graph



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 cluster?
- Choices:
  - TensorFlow
  - Theano
  - Torch (lua)

Deep Learning Libraries

- tf.learn (Tensorflow)
- Keras (Tensorflow, Theano)
- Lasagna (Theano)
- Torch.nn / PyTorch (torch)
- Chainer (chainer)
- MXNet (MXNet)

- Also see:
  http://mxnet.io/architecture/program_model.html

Introduction to tf.learn

Drop-out

Implementing Drop-Out

Experiments

Batch Normalization