Neural Networks -- two inputs, one hidden layer

In[1333]:=
```
x = 0; Remove["Global`*"]; DateList[Date[]] // Most
```

Out[1333]=
```
{2020, 5, 11, 23, 2}
```

We now examine a simple two-dimensional, single hidden layer, neural network.

Define neural net nodes and the approximation

G will be our neuron. Make G evaluate vectors elementwise

In[1334]:=
```
G[list_List] := G /@ list
```

We will first create a small neural net so that you can see the structure

In[1335]:=
```
numnodes = 3; dim = 2;
```

In[1336]:=
```
amat = Table[w_{i,j}, {i, 1, numnodes}, {j, 1, dim}];
bvec = Table[b_i, {i, 1, numnodes}];
basis = G /@ (amat. {x, y} + bvec)
```

Out[1338]=
$$\left\{G\left[b_1 + x\, w_{1,1} + y\, w_{1,2}\right], G\left[b_2 + x\, w_{2,1} + y\, w_{2,2}\right], G\left[b_3 + x\, w_{3,1} + y\, w_{3,2}\right]\right\}$$

In[1339]:=
```
cvec = Table[c_i, {i, Length[basis]}]
```

Out[1339]=
$$\{c_1, c_2, c_3\}$$

In[1340]:=
```
model = Sum[c_i basis[[i]], {i, Length[basis]}] + d_0
```

Out[1340]=
$$G\left[b_1 + x\, w_{1,1} + y\, w_{1,2}\right] c_1 + G\left[b_2 + x\, w_{2,1} + y\, w_{2,2}\right] c_2 + G\left[b_3 + x\, w_{3,1} + y\, w_{3,2}\right] c_3 + d_0$$

Math time

Mathematically, a neuron is equivalent to the function:

$$Y = \theta\left(\sum_{i=1}^{n} W_i X_i + b\right),$$

which can be conveniently modeled, using a matrix form,

$$Y = \theta(W.X + b),$$

where $W = [W_1 \quad W_2 \quad \cdots \quad W_n]$, and $X = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix}$.

A layer of neurons can be conveniently represented, using matrix notation, as follows:

$$W = \begin{bmatrix} W_{11} & \cdots & W_{1M} \\ \vdots & \vdots & \vdots \\ W_{N1} & \cdots & W_{NM} \end{bmatrix}.$$

The row index in each element of this matrix represents the destination neuron of the correspondi
connection, whereas the column index refers to the input source of the connection.

Designating by Y the output of the layer, you can write

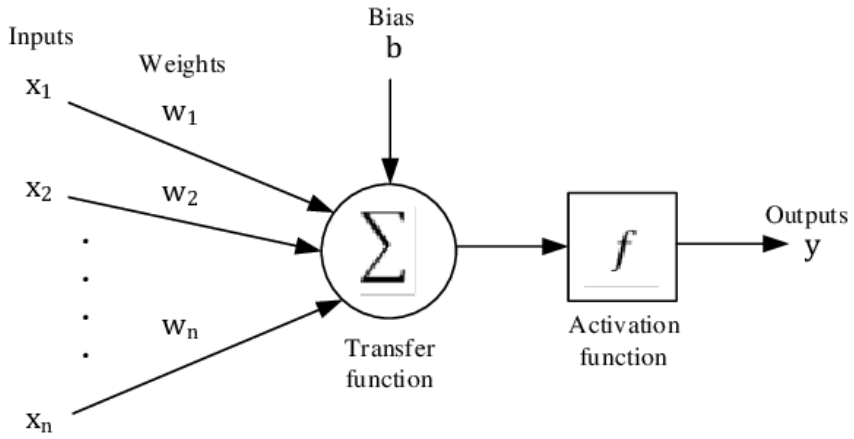
$$Y = \begin{bmatrix} Y_1 \\ \vdots \\ Y_i \\ \vdots \\ Y_N \end{bmatrix} = \begin{bmatrix} \theta\left(\sum_{j=1}^{M} W_{1j} X_j + b_1\right) \\ \vdots \\ \theta\left(\sum_{j=1}^{M} W_{ij} X_j + b_i\right) \\ \vdots \\ \theta\left(\sum_{j=1}^{M} W_{Nj} X_j + b_N\right) \end{bmatrix} = \theta\left(W.X + b\right),$$

The function achieved by this network is

$$\boldsymbol{Y}^3 = \begin{bmatrix} Y_1^3 \\ \vdots \\ Y_i^3 \\ \vdots \\ Y_{N3}^3 \end{bmatrix} = \theta\left(W^3 Y^2 + b^3\right) = \theta\left(W^3 \theta\left(W^2 Y^1 + b^2\right) + \boldsymbol{b}^3\right) = \theta\left(W^3 \theta\left(W^2\left(\theta\left(W^1 X + b^1\right)\right) + b^2\right) + b^3\right).$$

Picture time

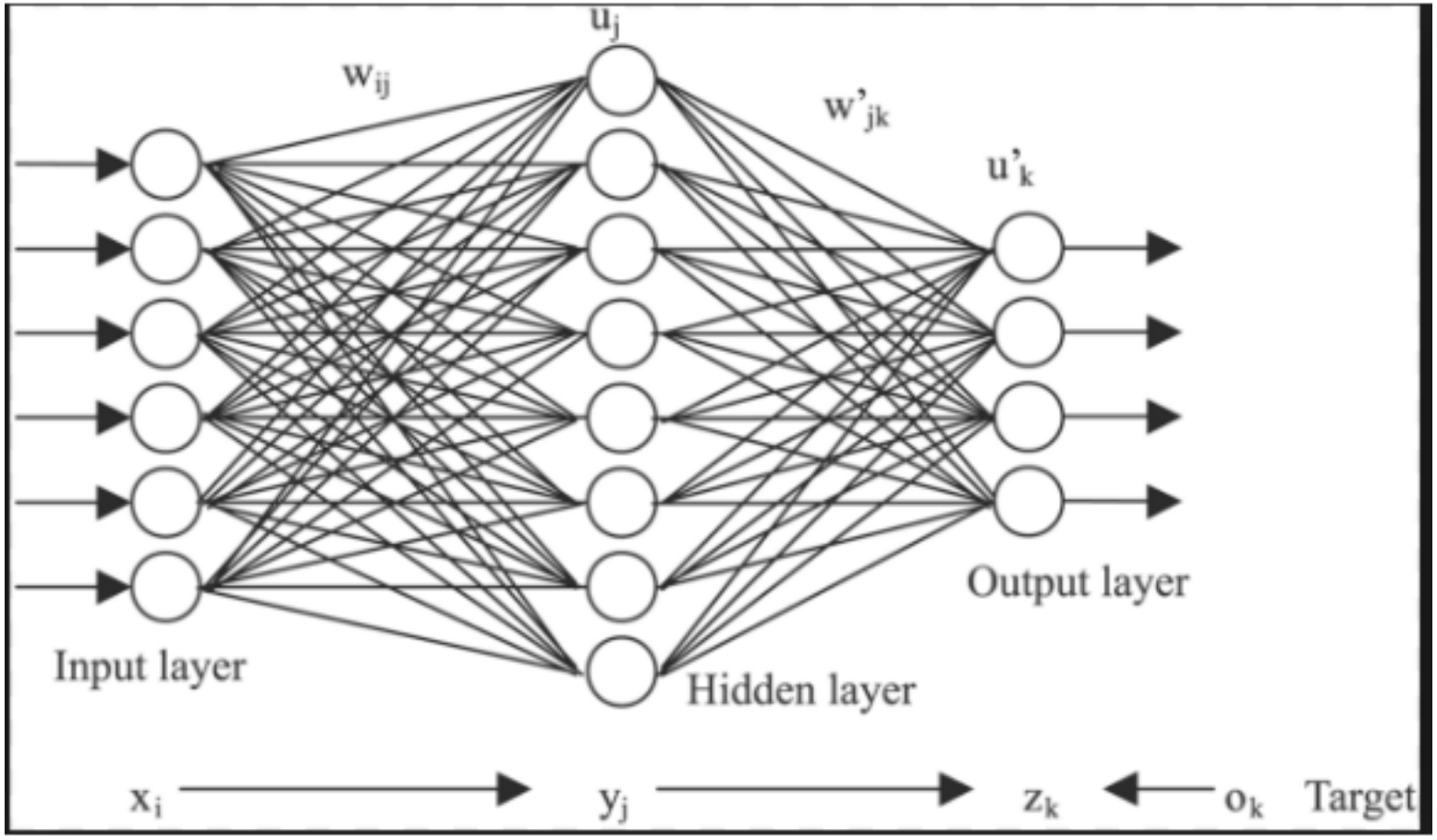
The neural net literature is filled with pictures, leaving the reader to decode the hieroglyphics in order to understand the underlying math. Here we see a common picture representing a single neuron with multiple inputs and a scalar output



Note the jargon:
Transfer function is nothing more than the weighted sum of the inputs plus the bias.
Bias is just an additive term
Activation function is just a function of one variable
"Outputs" should be "output" since it is a scalar for an individual neuron.

Fortunately, they got "inputs" right.

The picture above represents a single neuron. Here is what a full single hidden layer neural network looks like where each circle represents a neuron



We now specify a more serious net

In[1341]:=
```
numnodes = 9; dim = 2;
amat = Table[w_{i,j}, {i, 1, numnodes}, {j, 1, dim}];
bvec = Table[b_i, {i, 1, numnodes}];
basis = G /@ (amat. {x, y} + bvec);
cvec = Table[c_i, {i, Length[basis]}];
model = Sum[c_i basis[[i]], {i, Length[basis]}] + d0;
```

Define node function

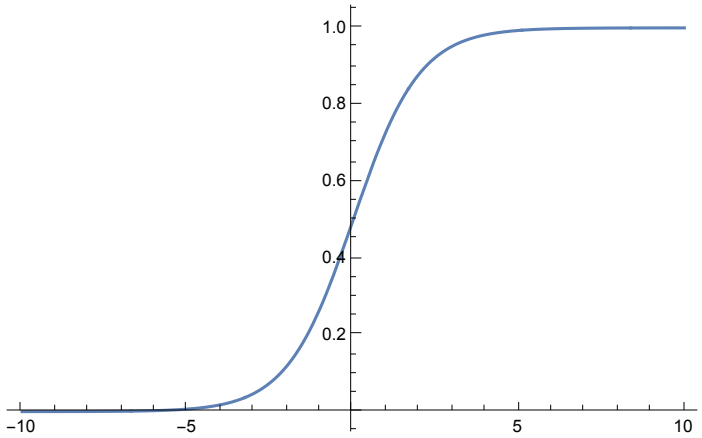
We use sigmoid function EXCEPT we avoid overflow and under flow

In[1347]:=
```
G[x_] = If[-20 ≤ x ≤ 20, 1 / (1 + Exp[-x]), If[x < -20, 0, 1]]
```

Out[1347]=
$$\text{If}\left[-20 \le x \le 20,\ \frac{1}{1+\text{Exp}[-x]},\ \text{If}[x < -20,\ 0,\ 1]\right]$$

In[1348]:=
```
Plot[G[x], {x, -10, 10}]
```

Out[1348]=



Set up the optimization

In[1349]:=
```
vars = Join[amat, bvec, cvec, {d0}] // Flatten
```

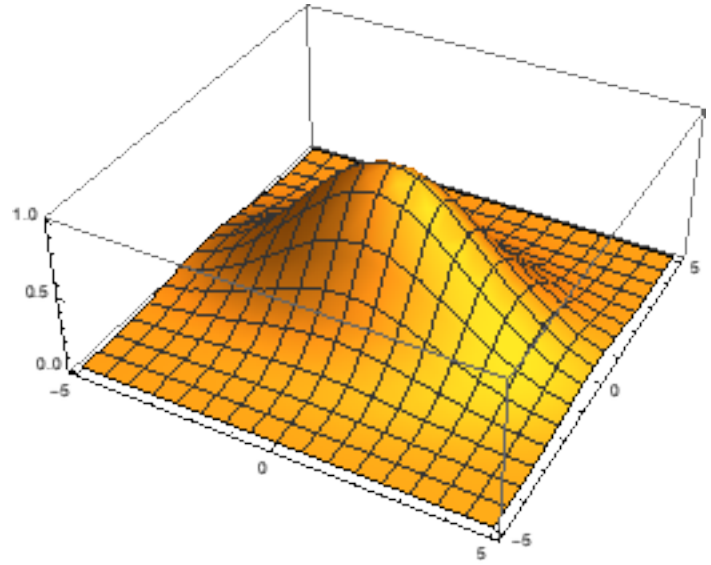
Out[1349]=
$$\{w_{1,1}, w_{1,2}, w_{2,1}, w_{2,2}, w_{3,1}, w_{3,2}, w_{4,1}, w_{4,2}, w_{5,1}, w_{5,2}, w_{6,1}, w_{6,2}, w_{7,1}, w_{7,2},$$
$$w_{8,1}, w_{8,2}, w_{9,1}, w_{9,2}, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, d0\}$$

In[1350]:=
```
init = vars - vars + 1;
varsin = {vars, init} // Transpose;
```

Fit a 2D example function

In[1352]:=
```
f[x_, y_] = Exp[-(x^2 / 10 + 2 y^2 / 8)];
Plot3D[f[x, y], {x, -5, 5}, {y, -5, 5}, PlotRange → All]
```

Out[1353]=

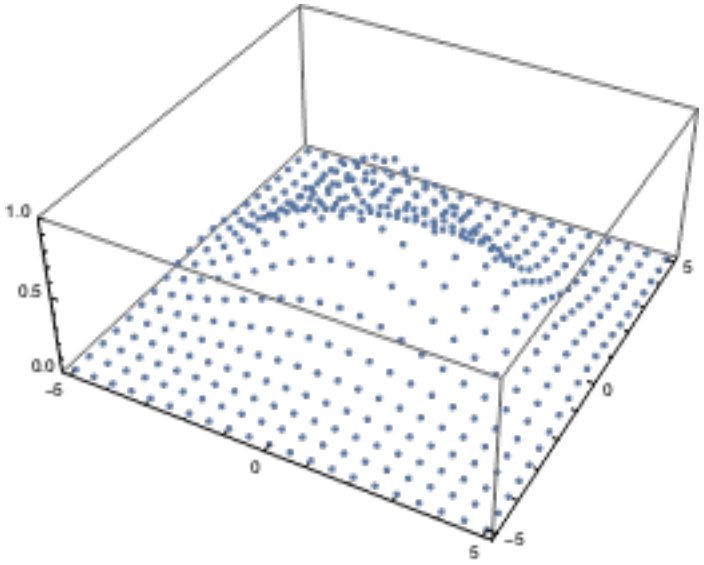


In[1354]:=
```
mpts = 10;
npts = 2 mpts + 1;
xpts = (Range[npts] - mpts - 1) / 2.
```

Out[1356]=
```
{-5., -4.5, -4., -3.5, -3., -2.5, -2., -1.5, -1., -0.5, 0., 0.5, 1., 1.5, 2., 2.5, 3., 3.5, 4., 4.5, 5.}
```

In[1357]:=
```
inputs = Flatten[Outer[List, xpts, xpts], 1];
inputsT = inputs // Transpose;
zpts = f @@@ inputs;
data = Join[inputsT, {zpts}];
dataT = data // Transpose;
ListPointPlot3D[dataT]
```

Out[1362]=



Let's use a global optimizer with multiple initial guesses to solve the least squares problem

In[1363]:=
```
fit = FindFit[dataT, model, varsin, {x, y}, MaxIterations → 100, Method -> "NMinimize"]
```

NMinimize: Failed to converge to the requested accuracy or precision within 100 iterations.

Out[1363]=
$$\{w_{1,1} \to -0.19068,\ w_{1,2} \to 0.790665,\ w_{2,1} \to 0.221148,\ w_{2,2} \to -0.929863,\ w_{3,1} \to 0.0568638,\ w_{3,2} \to 0.970384,\ w_{4,1} \to 0.222165,$$
$$w_{4,2} \to 1.18064,\ w_{5,1} \to 1.0348,\ w_{5,2} \to 0.69876,\ w_{6,1} \to 0.520327,\ w_{6,2} \to 0.921683,\ w_{7,1} \to 1.07408,\ w_{7,2} \to 0.0869792,$$
$$w_{8,1} \to 0.713561,\ w_{8,2} \to -0.911572,\ w_{9,1} \to -1.15364,\ w_{9,2} \to 0.506211,\ b_1 \to 0.341216,\ b_2 \to -0.644581,\ b_3 \to -3.02443,$$
$$b_4 \to 1.21553,\ b_5 \to 1.38668,\ b_6 \to -1.22828,\ b_7 \to -1.49811,\ b_8 \to -1.50161,\ b_9 \to -1.46302,\ c_1 \to -9.2217,\ c_2 \to -7.21993,$$
$$c_3 \to 0.956138,\ c_4 \to 1.25642,\ c_5 \to 0.436296,\ c_6 \to -1.07132,\ c_7 \to -0.452691,\ c_8 \to -0.738582,\ c_9 \to -0.375862,\ d0 \to 8.05017\}$$

Note:

Mathematica said it could not solve the problem to the standard 8-digit precision.

These problems are ill-conditioned, so we should use sloppy stopping rules.

Mathematica is using double precision arithmetic.

Most neural network software uses single precision.

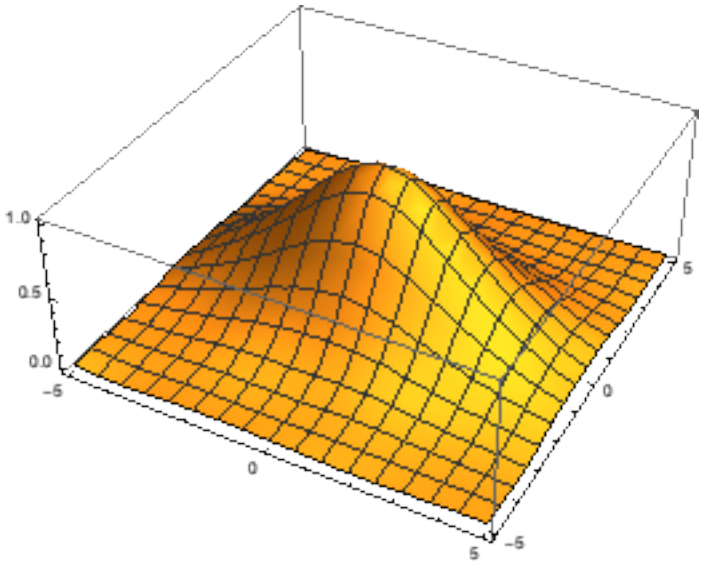
If you want better, you need to set the options

"Better" costs you time.

Plot the neural net result

In[1364]:=
```
modelf = Function[{x, y}, Evaluate[model /. fit]];
Plot3D[modelf[x, y], {x, -5, 5}, {y, -5, 5}]
```

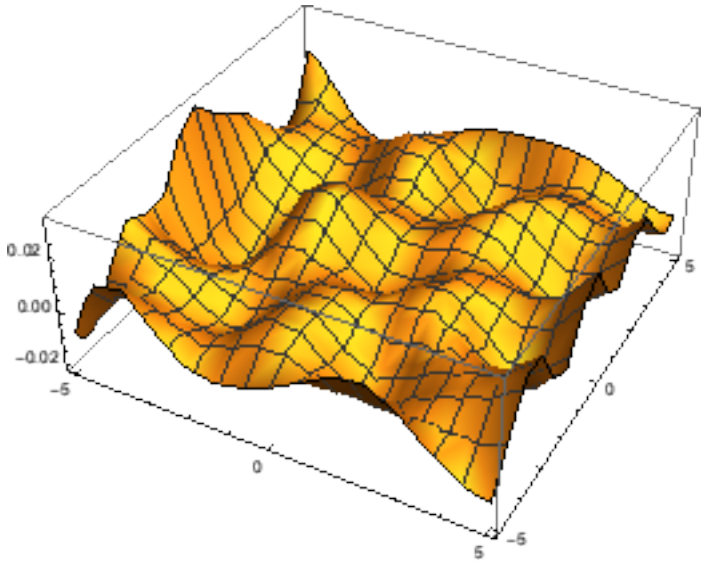
Out[1365]=



Plot the error. Note that the error is in [-0.02, 0.02].

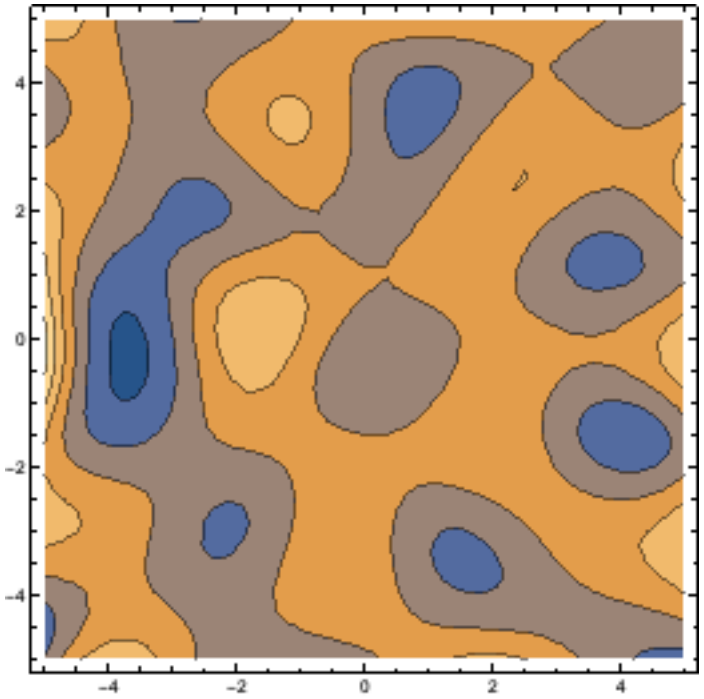
In[1366]:=
```
Plot3D[f[x, y] - modelf[x, y], {x, -5, 5}, {y, -5, 5}]
```

Out[1366]=



In[1367]:=
```
ContourPlot[f[x, y] - modelf[x, y], {x, -5, 5}, {y, -5, 5}]
```

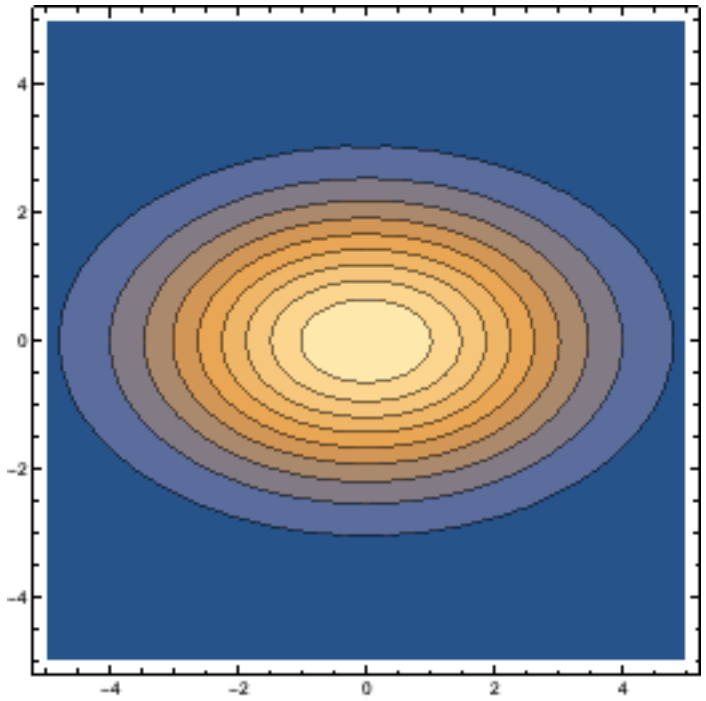
Out[1367]=



Let’s look at the contours of the true function

In[1368]:=
```
ContourPlot[f[x, y], {x, -5, 5}, {y, -5, 5}]
```

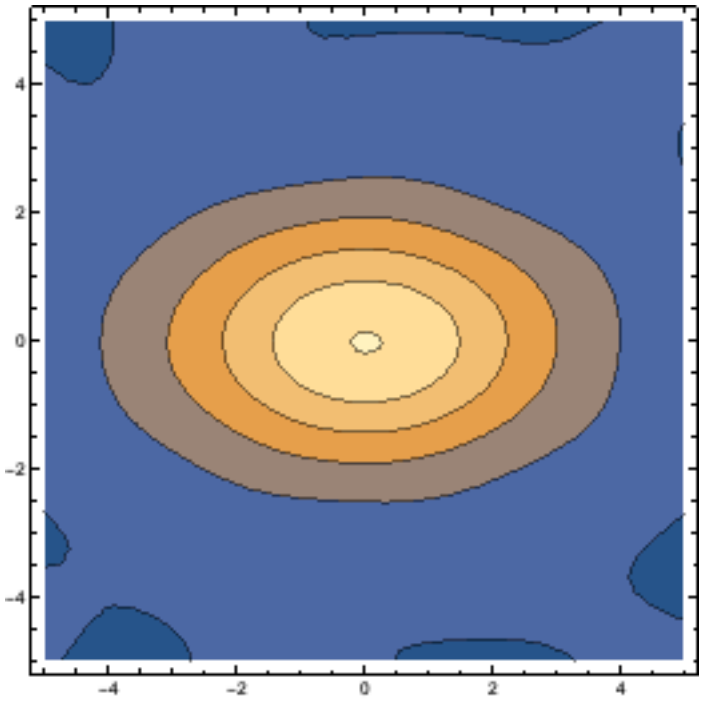
Out[1368]=



Here are the contours of the neural network

In[1369]:=
```
ContourPlot[modelf[x, y], {x, -5, 5}, {y, -5, 5}]
```

Out[1369]=



Comments on the optimization step

The optimization problems are hideously large.

- Standard methods are not used.
- The dominant method is stochastic gradient and its refinements.

Stochastic gradient

- Take steepest descent direction method

- Need the gradient: NNs are designed to make this easy!
  - They call it backpropagation; we call it automatic differentiation

- Computing gradient means evaluating NN for all input data
  - Too expensive!
  - Instead use a random sample

- Use this stochastically approximated gradient to adjust all weights and biases

Software and Hardware

TensorFlow is dominant neural net package

- The language looks horrible to me
- There are Python front ends to make it easier
- Challenge: You need to fight through the NN jargon

Massive parallelization is used.

It is the future

Becoming Human

Exploring Artificial Intelligence & What it Means to be Human