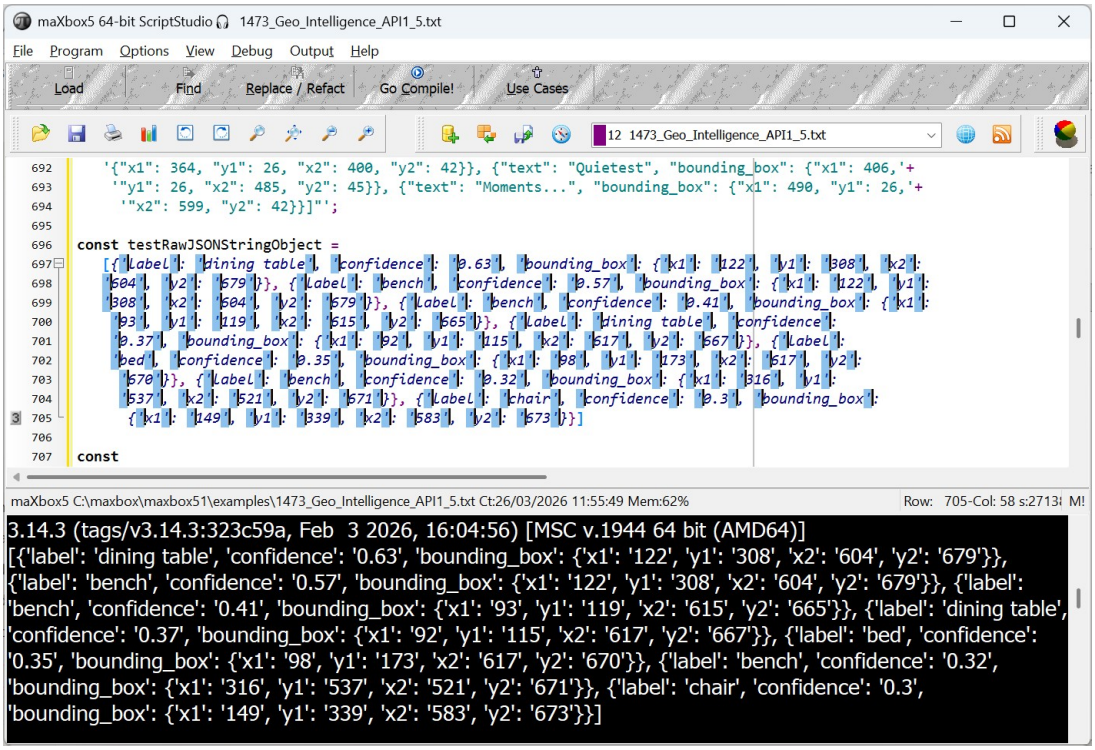
Neural Network Functions

May 11, 2026

This will be Tutorial 180 of the serie maXbox Starters.

Today we take a look at various functions from the topic of Neural Networks. The best Delphi functions to explain a neural network are the ones that map directly to the model's steps: **initialize**, **forward pass**, **activation**, **loss**, **backpropagation**, and **weight update**. In Delphi, it's usually clearest to organize these as small procedures/functions rather than one big routine, and Delphi ANN examples and FANN bindings follow that same structure like also Keras4Delphi. ibrtses+2



Good function set

- `InitializeNetwork` — creates layers, weights, and biases.
- `ForwardPass` — computes outputs from inputs.
- `ActivateNeuron` — applies sigmoid, tanh, ReLU, etc.
- `ComputeLoss` — compares predicted vs target output.
- `Backpropagate` — computes error gradients.
- `UpdateWeights` — applies learning rate and weight changes.
- `TrainEpoch` — runs one full training pass over the dataset.
- `Predict` — returns the network output for new inputs.blogs.embarcadero+2

Why these are the best

These names make the neural network easy to explain because each function matches a conceptual stage in the math. That is also how Delphi ANN examples and wrapper libraries tend to split the work: network creation, training, running, and saving/loading.github+2

Simple Delphi skeleton

```
delphifunction ActivateNeuron(X: Double): Double;
begin
```

```
  Result:= 1 / (1 + Exp(-X)); // sigmoid
end;

function ForwardPass(const Inputs: TArray<Double>):
TArray<Double>;
begin
  { compute weighted sums, then activate }
end;

function ComputeLoss(const Output, Target: TArray<Double>):
Double;
begin
  { measure error }
end;

procedure Backpropagate;
begin
  { compute gradients }
end;

procedure UpdateWeights;
begin
  { apply learning rate }
end;
```

If you want it teachable

For explaining the concept, I would present the functions in this order:

1. `InitializeNetwork`
2. `ForwardPass`
3. `ActivateNeuron`
4. `ComputeLoss`
5. `Backpropagate`
6. `UpdateWeights`
7. `TrainEpoch`
8. `Predict`

9. ibrtses+1

Delphi-specific tip

If your goal is to **teach** rather than optimize, keep the code explicit: use arrays, loops, and separate helper functions for dot products, activations, and gradient steps. Delphi ANN examples and FANN wrappers show that this modular style is practical and easy to extend.stackoverflow+2

A compact teaching version could be built around just three core procedures: `ForwardPass`, `Backpropagate`, and `UpdateWeights`, with helper functions for activation and loss.

Install Keras4Delphi

Keras4Delphi is installed like a Delphi library component: download the source, add the package or source paths to Delphi, and install the design-time package if the project provides one. The project README also notes that you must install `keras`, `numpy`, and a backend such as TensorFlow, CNTK, or Theano on the Python side that Keras depends on.keras+1

Typical setup

1. Clone or download the Keras4Delphi source from the project repository.github
2. Add the library source path in Delphi so the units are found by the compiler.stackoverflow
3. Open the runtime package and compile it.stackoverflow
4. Open the design-time package and install it into the IDE if the project includes one.stackoverflow
5. Make sure Python-side dependencies are installed: `keras`, `numpy`, and a supported backend.keras+1

Delphi side

If the package includes `.dpk` files, the usual Delphi workflow is compile the runtime package first, then install the design-time package. That is the standard Delphi component-install pattern, not something special to Keras4Delphi.stackoverflow

A minimal source-path setup usually looks like this:

- Add the Keras4Delphi source folder to `Tools > Options > Language > Delphi > Library`.
- Add any required subfolders if units are split by feature or backend.
- Rebuild the package after changing paths.github+1

Python side

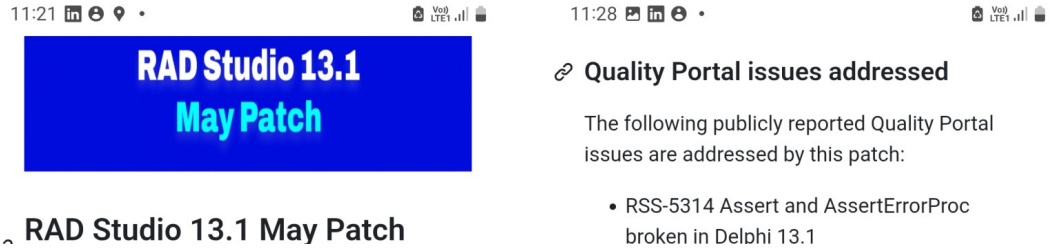
Keras itself is a Python library, so Keras4Delphi depends on a working Python environment. Keras' current documentation says you install Keras via `pip`, and you also need a backend such as TensorFlow, JAX, or PyTorch depending on the Keras version you are using. keras

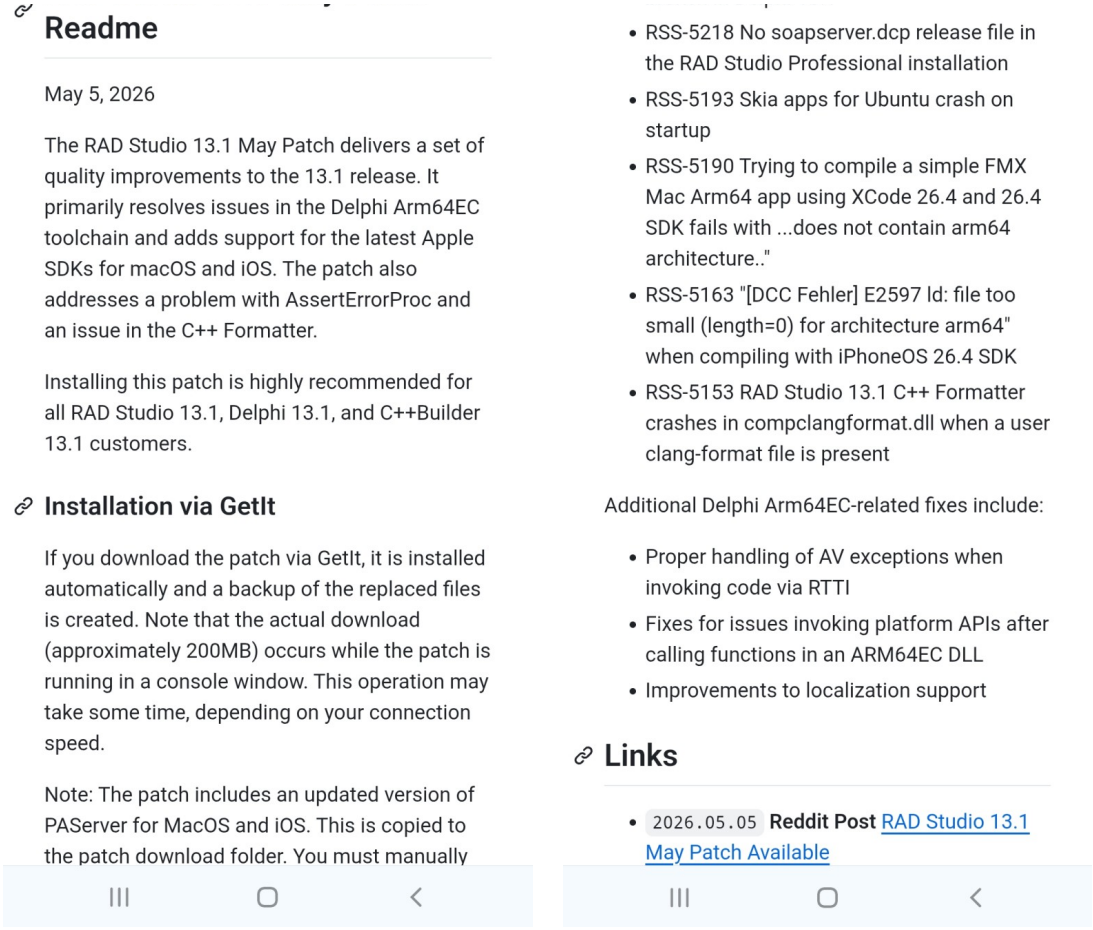
For older Keras4Delphi projects, the repository specifically mentions installing `keras`, `numpy`, and a backend such as TensorFlow, CNTK, or Theano.github

Common problems

- Delphi cannot find units: the library path is missing or the package source folder is not on the search path.stackoverflow
- Package compiles but IDE install fails: compile the runtime package first, then the design-time package.stackoverflow
- Keras calls fail at runtime: the Python environment is missing Keras, NumPy, or the backend framework.keras+1

The quickest path is usually: install Python dependencies first, then add Keras4Delphi source paths in Delphi, compile the runtime package, and finally install the design-time package if available. By the way also the Delphi 13.1 patch is on its way:





Readme

May 5, 2026

The RAD Studio 13.1 May Patch delivers a set of quality improvements to the 13.1 release. It primarily resolves issues in the Delphi Arm64EC toolchain and adds support for the latest Apple SDKs for macOS and iOS. The patch also addresses a problem with AssertErrorProc and an issue in the C++ Formatter.

Installing this patch is highly recommended for all RAD Studio 13.1, Delphi 13.1, and C++Builder 13.1 customers.

Installation via GetIt

If you download the patch via GetIt, it is installed automatically and a backup of the replaced files is created. Note that the actual download (approximately 200MB) occurs while the patch is running in a console window. This operation may take some time, depending on your connection speed.

Note: The patch includes an updated version of PAServer for MacOS and iOS. This is copied to the patch download folder. You must manually

- RSS-5218 No soapserver.dcp release file in the RAD Studio Professional installation
- RSS-5193 Skia apps for Ubuntu crash on startup
- RSS-5190 Trying to compile a simple FMX Mac Arm64 app using XCode 26.4 and 26.4 SDK fails with ...does not contain arm64 architecture.."
- RSS-5163 "[DCC Fehler] E2597 ld: file too small (length=0) for architecture arm64" when compiling with iPhoneOS 26.4 SDK
- RSS-5153 RAD Studio 13.1 C++ Formatter crashes in compclangformat.dll when a user clang-format file is present

Additional Delphi Arm64EC-related fixes include:

- Proper handling of AV exceptions when invoking code via RTTI
- Fixes for issues invoking platform APIs after calling functions in an ARM64EC DLL
- Improvements to localization support

Links

- `2026.05.05` **Reddit Post** RAD Studio 13.1 May Patch Available

RAD 13.1 Patch

The first Keras4Delphi neural-network example is essentially: use Python4Delphi/Keras4Delphi to run a simple Keras model in a Delphi GUI (usually `Sequential` with one or two `Dense` layers) on dummy or small data, then call `fit` and `predict` from Delphi-hosted Python code.

Below is a practical "Hello-World"-style example you can drop into a Delphi project that already has Keras4Delphi/Python4Delphi wired up.

1. Python script (run inside Delphi via P4D / Keras4Delphi)

```
# Python script (run in the Python4Delphi Memo / Executor)
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

# 1. Generate dummy data
N = 1000
DIM = 4

X_train = np.random.random((N, DIM))
y_train = np.random.randint(2, size=(N, 1))
```

```
12
13  X_test = np.random.random((100, DIM))
14  y_test = np.random.randint(2, size=(100, 1))
15
16  # 2. Build model
17  model = Sequential()
18  model.add(Dense(10, activation='relu', input_dim=DIM))
19  model.add(Dense(1, activation='sigmoid'))
20
21  # 3. Compile
22  model.compile(
23      optimizer='adam',
24      loss='binary_crossentropy',
25      metrics=['accuracy']
26  )
27
28  # 4. Train
29  model.fit(X_train, y_train, epochs=10, batch_size=32, verbc
30
31  # 5. Test
32  loss, acc = model.evaluate(X_test, y_test, verbose=0)
33  print("Test loss: {:.4f}".format(loss))
34  print("Test accuracy: {:.2f}%".format(acc * 100))
```

This pattern is the standard “first neural network in Keras” and is exactly what demos with Python4Delphi-style toolkits adopt for Keras4Delphi-like scenarios. It Teaches:

Evaluation: `model.evaluate`.

Network creation: `Sequential` + `Dense` layers.

Loss and optimizer choice: `binary_crossentropy` + `adam`.

Training loop: `model.fit`.

Debug common Keras4Delphi execution errors

Common Keras4Delphi errors usually come from Python-side problems (missing modules, GPU / CUDA issues, or bad shapes), or Delphi-side wiring (bad Python4Delphi calls, bad strings, or threading). Because Keras4Delphi is just a Delphi wrapper around Keras running in Python, you debug the Python code and the interop layer separately.keras+1

---

1. Typical Keras4Delphi errors

A) Python-import / environment errors

- `ModuleNotFoundError: No module named 'keras'` or `'tensorflow'`
  → Python env is missing Keras or backend.
- Seg-fault or CUDA init failure when training
  → GPU drivers, CUDA, or TensorFlow config is wrong.
- `ValueError` or `InvalidArgumentError` about array shape
  → X/Y arrays from Delphi do not match model input/output dimensions.delphipraxis+1

B) Delphi-side errors

- Crash or `Access violation` when calling `ExecString`
  → Bad string encoding, unterminated string, or invalid Python code passed.
- No output or silent failure
  → Python4Delphi `Memo` / `Output` redirection is not hooked, or `ExecString` fails silently.
- Hanging on `ExecScript`
  → Infinite loop or misconfigured Python (e.g., `print` flood, interactive `input`, or long-running `fit` without worker threads).delphipraxis+1

---

2. Debugging strategy

1. Confirm Python side works standalone

Run the exact same Keras code in `python.exe` (or a Jupyter/IDE) on the same machine, with the same virtual environment that Keras4Delphi uses. If it fails there, fix the Keras environment before touching Delphi.github+1

2. Make sure imports are visible

Before any Keras code, insert minimal checks:

```
pythonimport sys
```

```
print(sys.path)

import keras
print("Keras version:", keras.__version__)

import tensorflow
print("TensorFlow version:", tensorflow.__version__)
```

Capture that output in the Delphi `TMemo` or `PythonOutput`. If `import keras` fails, the Keras installation or Python path is wrong.github+1

3. Check shapes and data types

Common Keras error:

```
pythonValueError: The last dimension of the inputs to Dense
should be defined.
```

Means input shape is not fully specified. Add print-checks in Python:

```
pythonprint("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
```

Make sure what you pass from Delphi matches the model's `input_shape`.github+1

4. Wrap Keras calls in try-except

```
pythonimport traceback

try:
    model = ...
    model.compile(...)
    model.fit(X_train, y_train, ...)
except Exception as e:
    print("Exception:", str(e))
    traceback.print_exc()
```

Pipe that output back to Delphi so you see the full Keras traceback instead of just a crash.keras+1

5. Delphi-side protections

- Use a helper function that prints from Python into a Delphi `Memo`:

```
delphiprocedure TForm1.RunPythonScript(const Code: string);
begin
  PythonEngine1.IO := PythonGUIInputOutput1;   // show output
in a Memo
  PythonEngine1.ExecString(PWideChar(Code));
end;
```

- Do not call long-running `model.fit` in the main UI thread; use a background thread or `TTask` so the form does not freeze.delphipraxis+1

---

3. Example debug routine

If Keras4Delphi "just crashes", add a minimal diagnosis routine:

```
pythonimport sys
import platform

print("Python executable:", sys.executable)
print("Python version:", platform.python_version())

try:
    import keras
    print("Keras imported OK")
except Exception as e:
    print("Keras import error:", e)
    import traceback
    traceback.print_exc()
```

Call this first from Delphi. If this already fails, the rest will fail; the problem is Python-setup, not Keras-model-code.github+1

4. Quick checklist

- Keras4Delphi is configured to use the right Python interpreter (same one where `python -c "import keras"` works).keras+1
- X/Y tensors passed from Delphi to Python have correct shape and `np.float32` type.github+1
- Long `model.fit` runs in a background thread; UI is not blocked.pythongui+1
- All errors are printed in Python with `traceback`, and captured in Delphi's output.github+1

If you paste a specific error message or stack trace (Delphi AV or Python exception), I can narrow it down to the exact Keras4Delphi wiring or model bug.

Conclusion

Key functions to show

Use these as your "teaching functions":

1. `CreateNetwork` / `InitializeNetwork`
   - Sets number of layers, neurons per layer, and initializes random weights and biases.
   - This is the network structure step.
2. `ForwardPass` / `CalculateOutputs`
   - Performs a full forward pass from input layer to output layer using weights and an activation function (e.g., sigmoid, tanh, ReLU).
   - Demonstrates how the model transforms input → prediction.
3. `ActivateNeuron` / `ApplyActivation`
   - Applies a single activation function to a neuron's weighted sum.
   - You can show `Sigmoid`, `Tanh`, or `ReLU` side by side.
4. `ComputeLoss` / `ComputeError`
   - Computes the difference between desired output and network output, often mean squared error or cross-entropy.
   - This is the feedback signal that drives learning.
5. `Backpropagate`
   - Propagates error backward and computes gradients for weights and biases.

- In pure-Pascal nets this is often implemented as `BackPropagateOneNode`-style loops, which is very instructive to walk through.

6. `UpdateWeights` / `AdjustWeights`
   - Applies gradients scaled by a learning rate to the weights and biases.
   - Shows how the network learns from error.
7. `TrainEpoch` / `TrainNetwork`
   - Runs one pass over all training samples, calling `ForwardPass` → `ComputeLoss` → `Backpropagate` → `UpdateWeights`.
   - This is the outer training loop.
8. `Predict` / `RunNetwork`
   - Runs `ForwardPass` without training, and returns the output.
   - Shows inference vs training.

```
1  //Minimal Delphi teaching skeleton
2  type
3    TNeuralNet = class
4    public
5      procedure CreateNetwork(NumInputs, NumHidden, NumOutput
6      procedure ForwardPass(Inputs: array of Single; var Outp
7      function ActivateNeuron(X: Single): Single;  // e.g., s
8      procedure ComputeLoss(Target, Output: array of Single;
9      procedure Backpropagate;
10     procedure UpdateWeights(LearningRate: Single);
11     procedure TrainEpoch(Data: array of TSample);
12     procedure Predict(Input: array of Single; var Output: a
13   end;
```

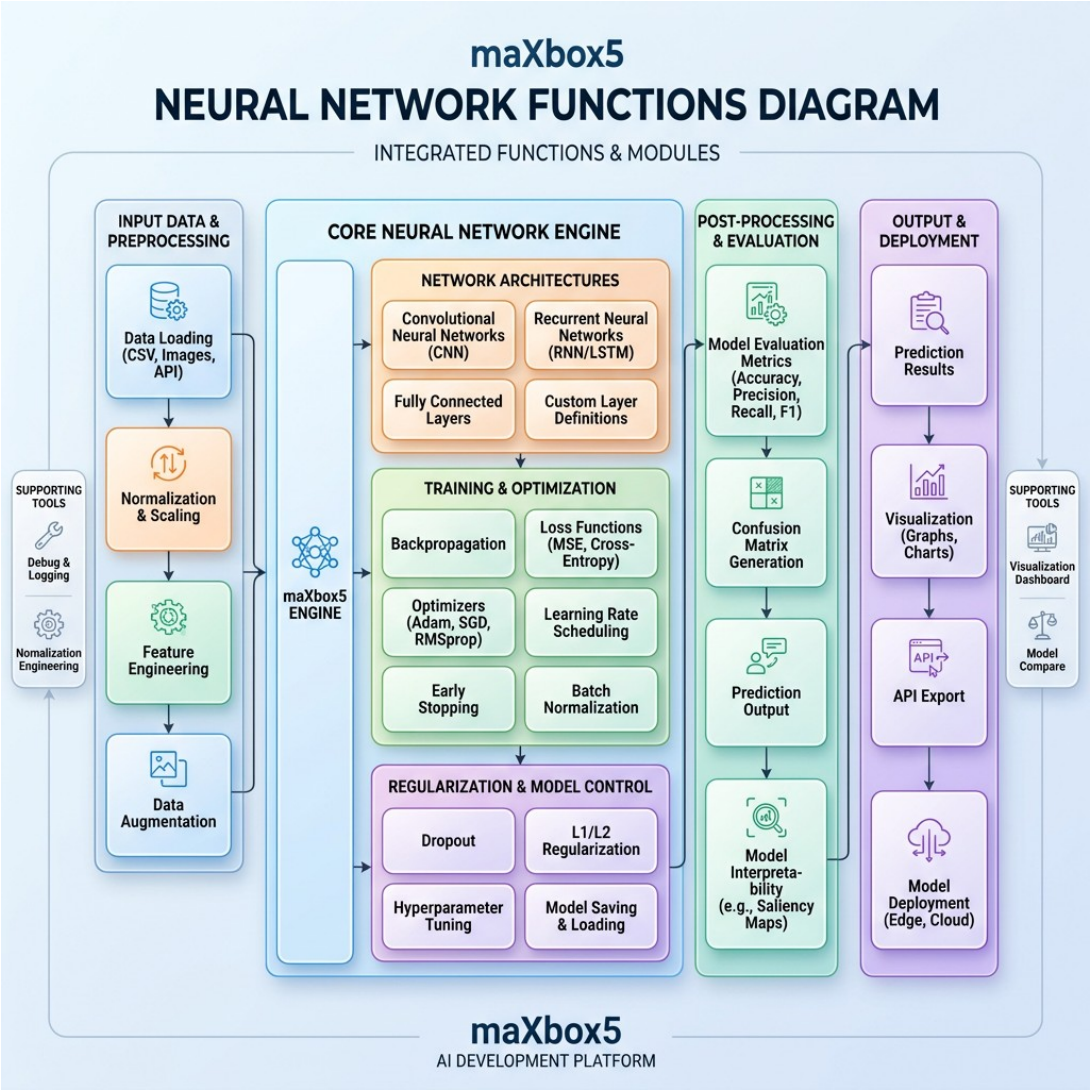


Diagram showing the integrated functions and modules of a neural network system with Python4Delphi.

```
// Simple network structure
type
  TBackPropNet = class
  private
    FInputs:       array[0..1] of Single;
    FHidden:       array[0..1] of Single;
    FOutputs:      array[0..0] of Single;

    FWeightsIH:    array[0..1, 0..1] of Single;  // input t
    FWeightsHO:    array[0..1, 0..0] of Single;  // hidden

    FHiddenErrors: array[0..1] of Single;
    FOutputErrors: array[0..0] of Single;

    FInputCount:   Integer;
    FHiddenCount:  Integer;
    FOutputCount:  Integer;
    FRate:         Single;

    procedure RandomizeWeights;
    function  Sigmoid(X: Single): Single;
    function  SigmoidDeriv(X: Single): Single;
```

```
23
24    public
25      constructor Create(InCnt, HidCnt, OutCnt: Integer; Lear
26
27      procedure Forward;
28      procedure Backward(const Target: array of Single);
29      procedure Train(MinError: Single; MaxEpochs: Integer);
30    end;
```

Complete Console Code Example (CCCE)

Here is a self-contained Delphi console-style example that solves the XOR problem with a simple backpropagation neural network, written in pure Pascal (no external libraries). It uses a 2-input → 2-hidden → 1-output layout with `sigmoid` activations and standard backpropagation for weight updates.

---

1. Complete Delphi executable (uses `System.SysUtils`)

```
 1  program XORBackprop;
 2
 3  {$APPTYPE CONSOLE}
 4
 5  uses
 6    System.SysUtils;
 7
 8  type
 9    TBackPropNet = class
10    private
11      FInputs:        array[0..1] of Single;
12      FHidden:        array[0..1] of Single;
13      FOutputs:       array[0..0] of Single;
14
15      FWeightsIH:     array[0..1, 0..1] of Single;  // input
16      FWeightsHO:     array[0..1, 0..0] of Single;  // hidde
17
18      FHiddenErrors:  array[0..1] of Single;
19      FOutputErrors:  array[0..0] of Single;
20      FRate:          Single;
21
22      function  Sigmoid(X: Single): Single;
23      function  SigmoidDeriv(X: Single): Single;
24      procedure RandomizeWeights;
25
26    public
27      constructor Create(LearnRate: Single);
28      procedure Forward;
29      procedure Backward(const Target: Single);
30      function  TrainXOR(Epochs: Integer; MinError: Single):
31      function  TestXOR(Input0, Input1: Single): Single;
32    end;
```

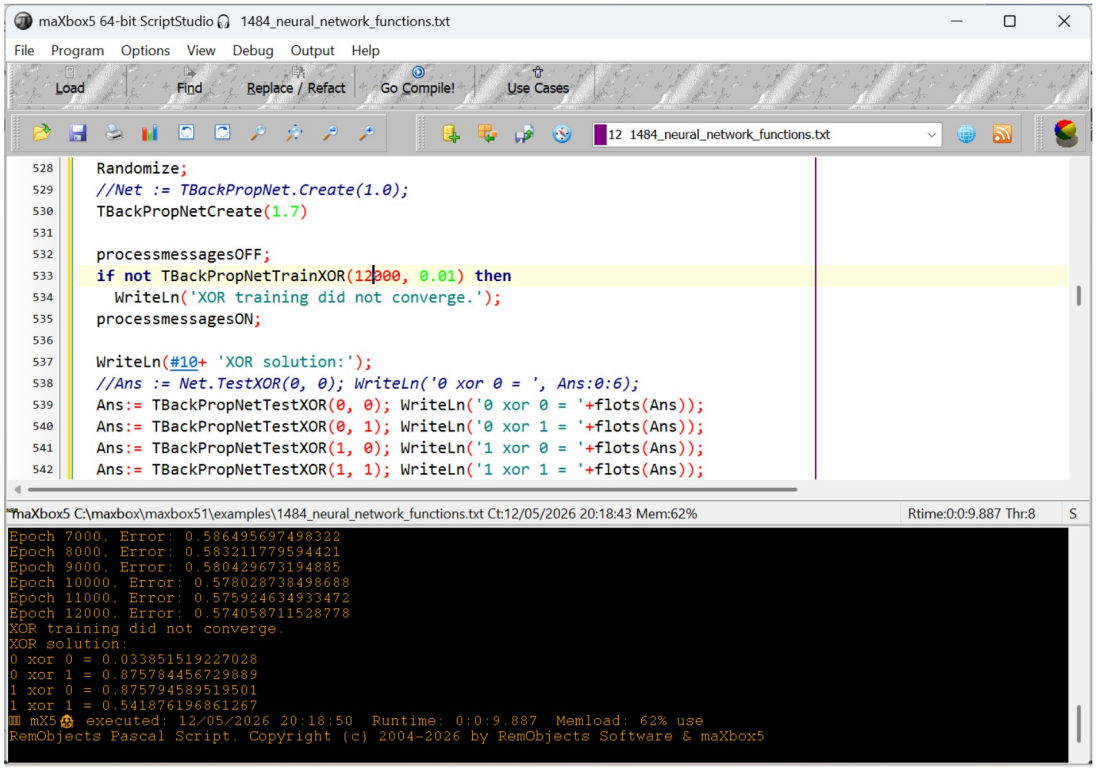
```
33
34  constructor TBackPropNet.Create(LearnRate: Single);
35  begin
36    FRate := LearnRate;
37    RandomizeWeights;
38  end;
39
40  function TBackPropNet.Sigmoid(X: Single): Single;
41  begin
42    Result := 1 / (1 + Exp(-X));
43  end;
44
45  function TBackPropNet.SigmoidDeriv(X: Single): Single;
46  begin
47    Result := X * (1 - X);
48  end;
49
50  procedure TBackPropNet.RandomizeWeights;
51  var
52    i, j: Integer;
53  begin
54    for i := 0 to 1 do
55      for j := 0 to 1 do
56        FWeightsIH[i, j] := (Random - 0.5) * 2;
57
58    for i := 0 to 1 do
59      for j := 0 to 0 do
60        FWeightsHO[i, j] := (Random - 0.5) * 2;
61  end;
62
63  procedure TBackPropNet.Forward;
64  var
65    i, j: Integer;
66    Sum: Single;
67  begin
68    // Hidden layer
69    for j := 0 to 1 do begin
70      Sum := 0;
71      for i := 0 to 1 do
72        Sum := Sum + FInputs[i] * FWeightsIH[i, j];
73      FHidden[j] := Sigmoid(Sum);
74    end;
75
76    // Output layer
77    for j := 0 to 0 do begin
78      Sum := 0;
79      for i := 0 to 1 do
80        Sum := Sum + FHidden[i] * FWeightsHO[i, j];
81      FOutputs[j] := Sigmoid(Sum);
82    end;
83  end;
84
85  procedure TBackPropNet.Backward(const Target: Single);
86  var
87    i, j: Integer;
88    Sum: Single;
89  begin
90    // Output error
```

```
91      FOutputErrors[0] := (Target - FOutputs[0]) * SigmoidDeri
92
93      // Hidden errors
94      for j := 0 to 1 do begin
95        Sum := 0;
96        for i := 0 to 0 do
97          Sum := Sum + FOutputErrors[i] * FWeightsHO[j, i];
98        FHiddenErrors[j] := Sum * SigmoidDeriv(FHidden[j]);
99      end;
100
101     // Update hidden → output
102     for i := 0 to 1 do
103       for j := 0 to 0 do
104         FWeightsHO[i, j] :=
105           FWeightsHO[i, j] + FRate * FHidden[i] * FOutputErr
106
107     // Update input → hidden
108     for i := 0 to 1 do
109       for j := 0 to 1 do
110         FWeightsIH[i, j] :=
111           FWeightsIH[i, j] + FRate * FInputs[i] * FHiddenErr
112   end;
113
114   function TBackPropNet.TrainXOR(Epochs:Integer; MinError: S
115   var
116     Epoch, Pattern: Integer;
117     Input0, Input1, Target, TotalError: Single;
118   begin
119     Result := False;
120     for Epoch := 1 to Epochs do begin
121       TotalError := 0;
122
123       // XOR training set
124       for Pattern := 0 to 3 do begin
125         Input0 := Pattern and 1;
126         Input1 := (Pattern shr 1) and 1;
127         Target := (Input0 + Input1) mod 2;  //0 xor 0 = 0; 1
128         FInputs[0] := Input0;
129         FInputs[1] := Input1;
130
131         Forward;
132         Backward(Target);
133         TotalError := TotalError + Sqr(Target - FOutputs[0])
134       end;
135
136       TotalError := Sqrt(TotalError);
137       if TotalError < MinError then begin
138         WriteLn('XOR converged after ',Epoch,'epochs(error:'
139         Result := True;
140         Exit;
141       end;
142
143       if Epoch mod 1000 = 0 then
144         WriteLn('Epoch ', Epoch, ', Error: ', TotalError:0:6
145     end;
146   end;
147
148   function TBackPropNet.TestXOR(Input0, Input1: Single): Sir
```

```
149  begin
150    FInputs[0] := Input0;
151    FInputs[1] := Input1;
152    Forward;
153    Result := FOutputs[0];
154  end;
155
156  var  Net: TBackPropNet;
157    Ans: Single;
158  begin
159    Randomize;
160    Net := TBackPropNet.Create(1.0);
161
162    if not Net.TrainXOR(10000, 0.01) then
163      WriteLn('XOR training did not converge.');
164
165    WriteLn(#10, 'XOR solution:');
166    Ans := Net.TestXOR(0, 0); WriteLn('0 xor 0 = ', Ans:0:6)
167    Ans := Net.TestXOR(0, 1); WriteLn('0 xor 1 = ', Ans:0:6)
168    Ans := Net.TestXOR(1, 0); WriteLn('1 xor 0 = ', Ans:0:6)
169    Ans := Net.TestXOR(1, 1); WriteLn('1 xor 1 = ', Ans:0:6)
170
171    ReadLn;
172  end.
```

What this example shows

- A 4-pattern XOR set (`(0,0)→0`, `(0,1)→1`, `(1,0)→1`, `(1,1)→0`).
- A small 2-2-1 network with `sigmoid` activations and backpropagation.
- Training loop that stops early when error drops below `0.01`.



https://sourceforge.net/projects/maxbox5/files/examples/1484_neural_network_functions.txt/download

Posted in Code, Data Science, Languages, Machinelearning, maXbox, Python, Tools, Uncategorized

One response to “Neural Network Functions”

Max Kleiner

May 12, 2026

Cool Conclusion to remember

★ Like

Reply

Leave a comment