ELEC 305

Digital System Design Lab

Fall 2024

Lecture 8:

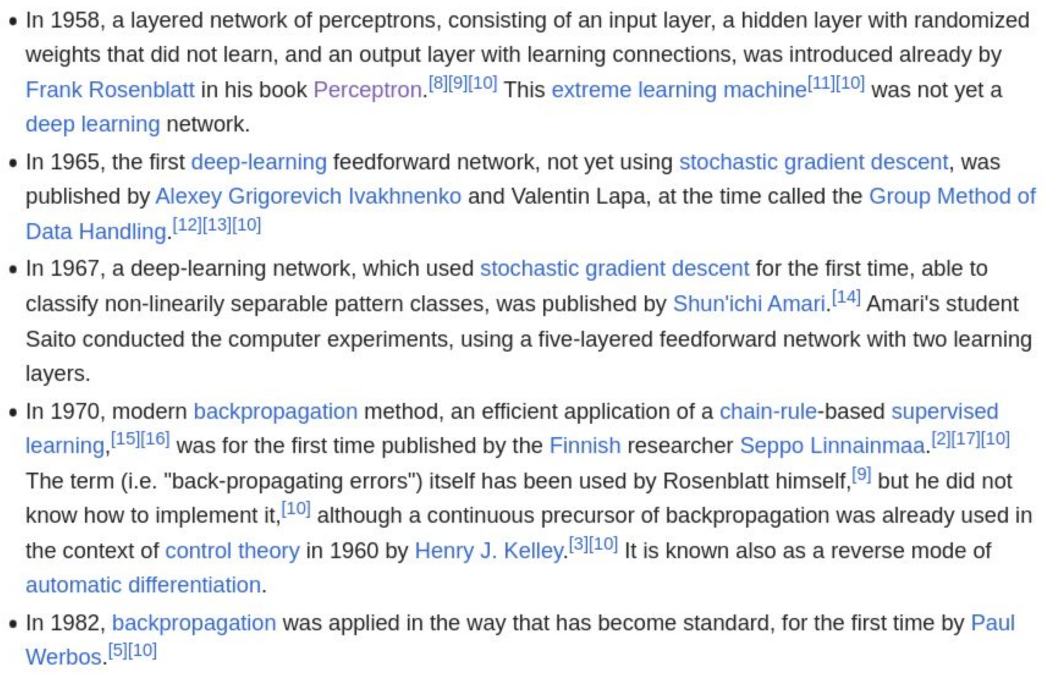
Neural Networks - Short Introduction and FPGA Implementation

Outline

- Neural Networks (NNs): Definitions and some history
- MLP vs. CNN: Different types of NNs
- NN Inference on FPGAs
  - Types of operators in NNs inference
  - Quantization (and fixed point numbers) in NN inference
  - Parallelizability

Neural Networks

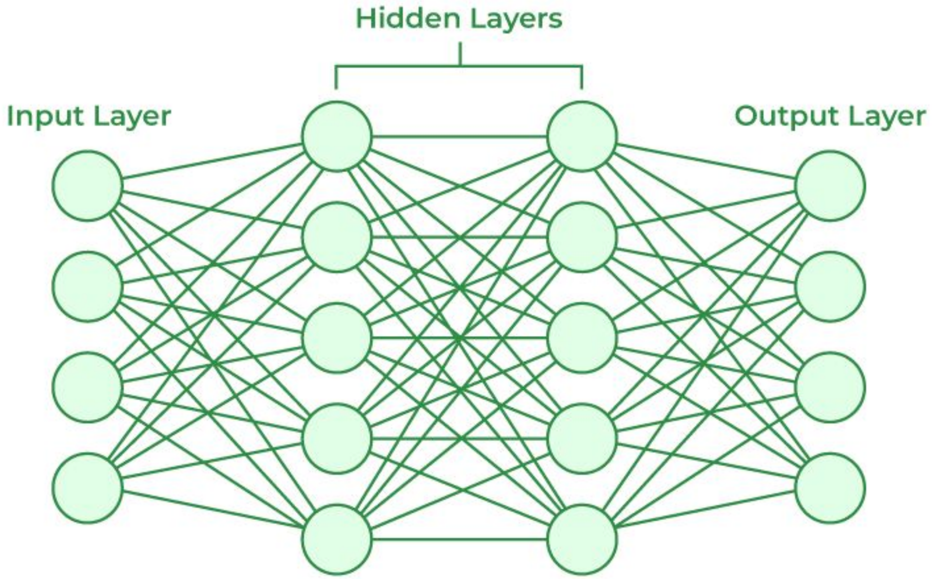
- A class of optimizable algorithms that were recently shown to be very powerful for **many** tasks
- They are not new! Fundamental results since the 1960s
- Optimizable = you don't explicitly program it to do a task, you have example data for that task + a "loss" function that shows you how well the task is being done, the neural net **learns** by using these



- In 1958, a layered network of perceptrons, consisting of an input layer, a hidden layer with randomized weights that did not learn, and an output layer with learning connections, was introduced already by Frank Rosenblatt in his book Perceptron.[8][9][10] This extreme learning machine[11][10] was not yet a deep learning network.
- In 1965, the first deep-learning feedforward network, not yet using stochastic gradient descent, was published by Alexey Grigorevich Ivakhnenko and Valentin Lapa, at the time called the Group Method of Data Handling.[12][13][10]
- In 1967, a deep-learning network, which used stochastic gradient descent for the first time, able to classify non-linearily separable pattern classes, was published by Shun'ichi Amari.[14] Amari's student Saito conducted the computer experiments, using a five-layered feedforward network with two learning layers.
- In 1970, modern backpropagation method, an efficient application of a chain-rule-based supervised learning,[15][16] was for the first time published by the Finnish researcher Seppo Linnainmaa.[2][17][10] The term (i.e. "back-propagating errors") itself has been used by Rosenblatt himself,[9] but he did not know how to implement it,[10] although a continuous precursor of backpropagation was already used in the context of control theory in 1960 by Henry J. Kelley.[3][10] It is known also as a reverse mode of automatic differentiation.
- In 1982, backpropagation was applied in the way that has become standard, for the first time by Paul Werbos.[5][10]

Neural Networks

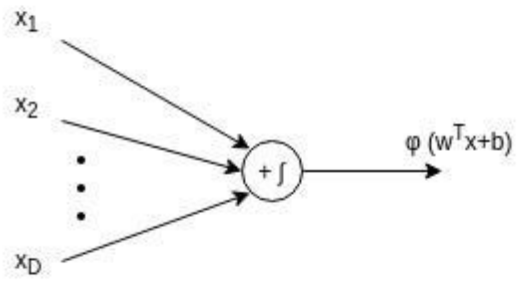
- The simplest NN is called a “multi layer perceptron”: MLP
- Let’s define NN structure based on it
- We have an input vector of size S1, the MLP will generate an output vector of size S2
- To generate this output, the MLP will apply a series of parametrized transformations on the data (e.g., matmul with “weights”, add with “biases”, …)



img src: https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/

Neural Networks

- These transformations are basically one unit repeating itself over and over again
- The unit is typically called a “neuron”
- Neuron output = weighted sum of all input elements + a bias term, passed through a nonlinear “activation function”
- Many neurons come together to form a “layer”
- Each layer has different weights and bias values, and sometimes also different activation fcns.

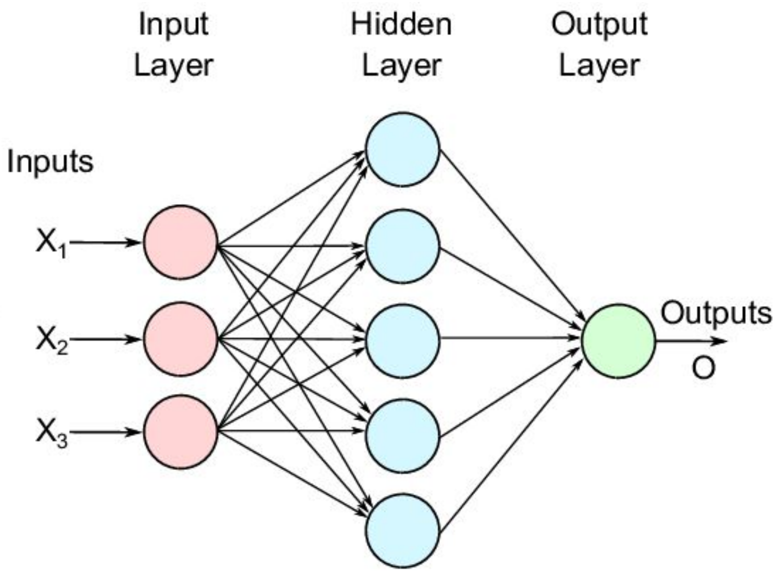


img src: https://medium.com/@AI_with_Kain/understanding-of-multilayer-perceptron-mlp-8f179c4a135f

- You actually know this form…
  A single neuron with a sigmoid activation is logistic regression!
- No activation = linear regression!

Neural Networks

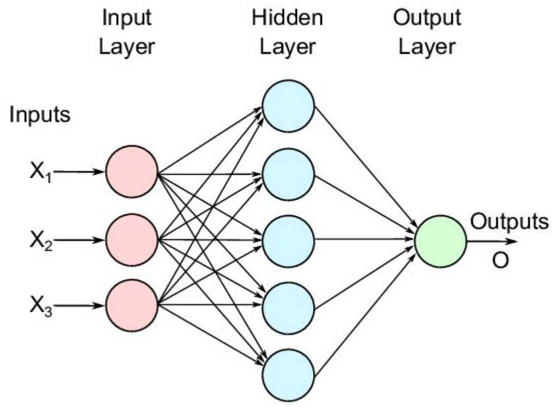
- A single neuron can only do so much though
- The power of MLP (and arguably, NNs in general) comes from cascading these neurons together
- Making layers “wider”, and networks of layers “deeper” (buzzword alert: deep learning)
- When this is done, some layers stay “inside” → these are called “hidden” layers
- As the number of hidden layers increase, you get deeper networks



img src: https://www.researchgate.net/publication/304701350_A_New_Multi-layer_Perceptrons_Trainer_Based_on_Ant_Lion_Optimization_Algorithm/figures?lo=1&utm_source=google&utm_medium=organic

Neural Networks

- E.g., this MLP takes an input vector x of size 3
- Multiplies each x element with a different weight
  - 3 weights and 1 bias for each neuron, 4 neurons in total:
    - $\{w_{11}, w_{12}, w_{13}\}, \{w_{21}, w_{22}, w_{23}\}, \{w_{31}, w_{32}, w_{33}\}, \{w_{41}, w_{42}, w_{43}\}$
    - $\{b_{o1}\}, \{b_{o2}\}, \{b_{o3}\}, \{b_{o4}\}$
- Does a per-neuron addition of the 3 mult results
- Feeds each neuron sum through an activation fcn
- Multiplies each activation with a different weight:
  - 4 weights and 1 bias in total: $w_{o1}, w_{o2}, w_{o3}, w_{o4}, b_o$



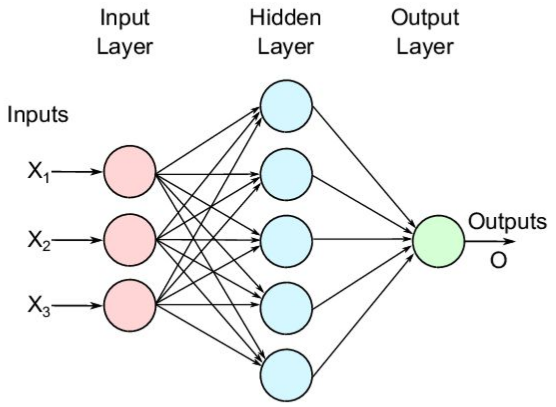
- Sums the output of the $w_{o\#}$ mult and declares that as the network output
  - Output is a scalar, or 1-elem vector

Neural Networks

- So it's actually possible to write this whole MLP down as one simple equation:

Let φ be the nonlinear activation function

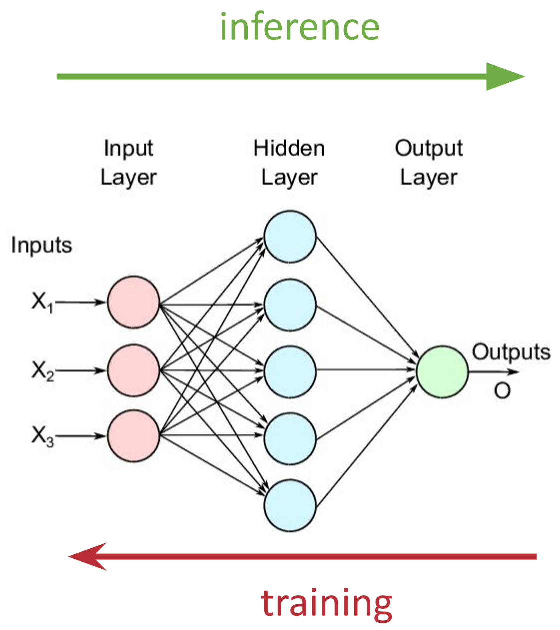
$$
\begin{aligned}
O = {} & w_{o1} * \varphi(x_1*w_{11} + x_2*w_{12} + x_3*w_{13} + b_{o1}) + \\
& w_{o2} * \varphi(x_1*w_{21} + x_2*w_{22} + x_3*w_{23} + b_{o2}) + \\
& w_{o3} * \varphi(x_1*w_{31} + x_2*w_{32} + x_3*w_{33} + b_{o3}) + \\
& w_{o4} * \varphi(x_1*w_{41} + x_2*w_{42} + x_3*w_{43} + b_{o4}) + b_o
\end{aligned}
$$



- Not the shortest equation ever, but it's surely simple, just adds and mults + one nonlinear function (like the sigmoid)

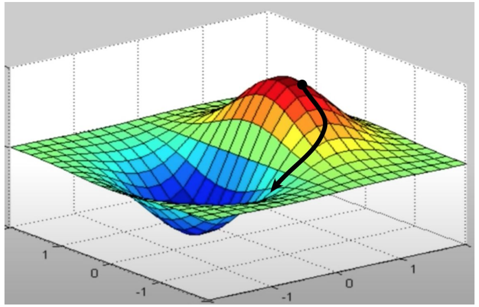
Neural Networks

- Networks are “trained” = parameters (weights, biases) are updated based on a specific optimization algorithm
- Most opt. algo.s used today are flavors of “gradient descent”:
  - Feed a certain # of x examples through the network, end up with certain y output predictions from the network.
  - Compare predictions with the ground truth for those y values (“labels”) via your loss function (opt. objective)
  - Compute the partial gradient from the loss to each parameter $\longrightarrow \delta(\text{loss}) / \delta(w_{ij})$

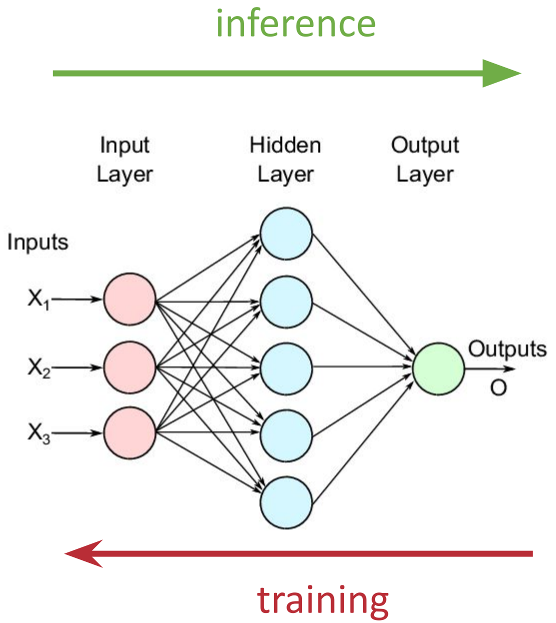


Neural Networks

- On a high dimensional manifold (dim = # of parameters), gradients tell you which direction you need to update your weights (+ or - ?) to make the loss lower (i.e., your output more correct).
- This is training.
- We won't do any training, this was just FYI
- We'll only do inference on trained networks, inference considers the MLP equation we wrote out earlier



img src: https://mriquestions.com/back-propagation.html



Neural Networks

- This single layer MLP has an interesting theoretical property:
- Given enough parameters and proper activation functions, the single-hidden-layer MLP can approximate **any** bounded and continuous function to an arbitrary error level (Cybenko, 1989)
  - How many parameters and which activation functions you need for a certain approximation error level, for a given task, is very hard to determine though (I'm not sure it's even possible for an arbitrary error level)
- This is huge though!! $\rightarrow$ it basically means if you have enough data to represent your task well, a "correctly-built" MLP, and a suitable optimization algorithm, your MLP will be capable of solving any task you throw at it

Neural Networks

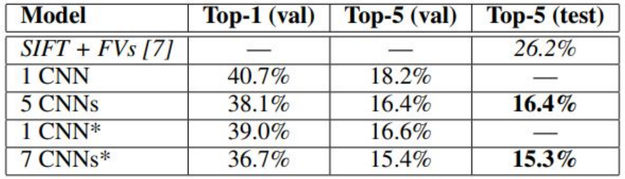
- Limitations:
  - Hard to find how much data is “enough” for any non-trivial task (see GPT-x studies)
  - Hard to find “correctly-built” MLP, how many parameters do we need?
  - Hard to find an optimization algorithm that allows for finding the correct parameters
- So it’s hard in all aspects 🙃, especially if you don’t have enough computational power
- It’s safe to say this is what caused the “AI Winters” of the past, and why the current AI boom went much further than the earlier ones

Neural Networks

- In summary, the hypotheses were there
  - *“Actually if we were able to train this huge model on lots of data, this would probably work”*
- But there wasn’t enough computational power or data for many of the challenging tasks
- One challenging task was image classification ⟶ the ILSVRC competition ranked methods against each other.
- Up to 2012 people had engineered algorithms that were winning this competition (SIFT / SURF features etc.)

Neural Networks

- Come 2012 → Alexnet (Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton)
- CUDA (the language used for programming NVIDIA GPUs) was predominantly used for signal processing and scientific computing before this paper
- The paper presented a CUDA implementation of CNNs that allowed orders of magnitude faster training, allowing the models to be trained "further"
- Alexnet killed its competition by a large margin, kicking off the AI revolution we know of today
- Well we saw MLPs, what's a CNN though?

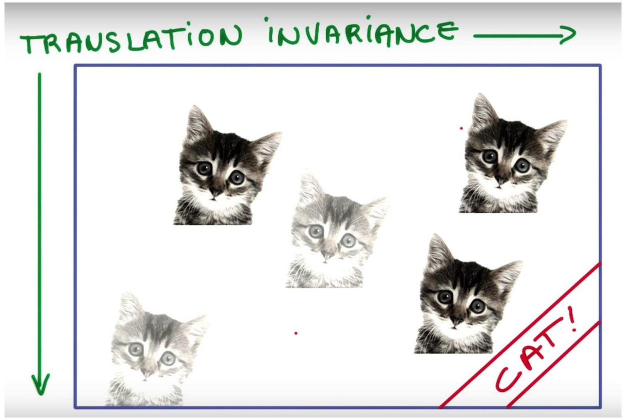


Model	Top-1 (val)	Top-5 (val)	Top-5 (test)
SIFT + FVs [7]	—	—	*26.2%*
1 CNN	40.7%	18.2%	—
5 CNNs	38.1%	16.4%	**16.4%**
1 CNN*	39.0%	16.6%	—
7 CNNs*	36.7%	15.4%	**15.3%**

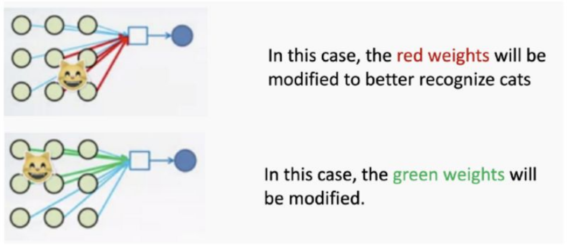
Table 2: Comparison of error rates on ILSVRC-2012 validation and test sets. In *italics* are best results achieved by others. Models with an asterisk* were "pre-trained" to classify the entire ImageNet 2011 Fall release. See Section 6 for details.

Neural Network Types

- One problem MLPs have is that they work on fixed-size inputs, i.e., x and y size are fixed
- This is a problem for image or audio processing tasks, where you can have data points of virtually any size
- Also, the MLP is not translation-invariant:
  - If MLP is trained with cats appearing on top left side of the image (say, from $x_{1,1}$ to $x_{102,57}$), it will not recognize cats appearing anywhere else
  - This is not desirable for most tasks!



img src: https://aiplanet.com/learn/getting-started-with-deep-learning/convolutional-neural-networks/267/cnn-transfer-learning-data-augmentation


In this case, the red weights will be modified to better recognize cats

In this case, the green weights will be modified.

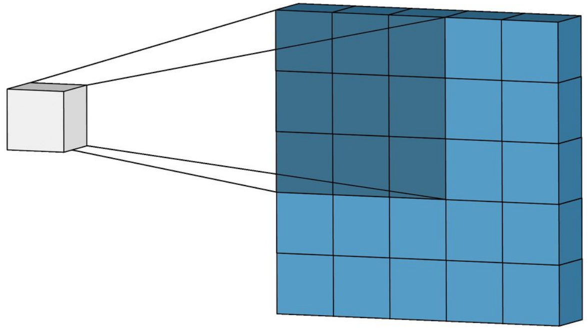
A cat detector using an MLP which changes as the position of the cat changes.

Neural Network Types: CNN

- We already know of a translation-invariant linear transformation: convolution

- FIR filters (applied via convolution) slide over the input data that needs to be filtered, and apply the same operation over and over again to all parts.

- Using this instead of the matrix multiplication in MLP (weight multiplication is actually matmul if you think about it) gives us translation-invariant operation.



- One disclaimer: the deep learning people don't usually care too much about DSP, and since filter parameters are learned here (rather than set w.r.t. some rules like the ones in FIIIR.com), you will see absolutely brazen ("biblically accurate" if you will) versions of convolutions around ([dilated grouped convs]() etc.), don't let those confuse you, our only relation to "convolving" here is this translation-invariance property, the rest is AI engineering.

Neural Network Types

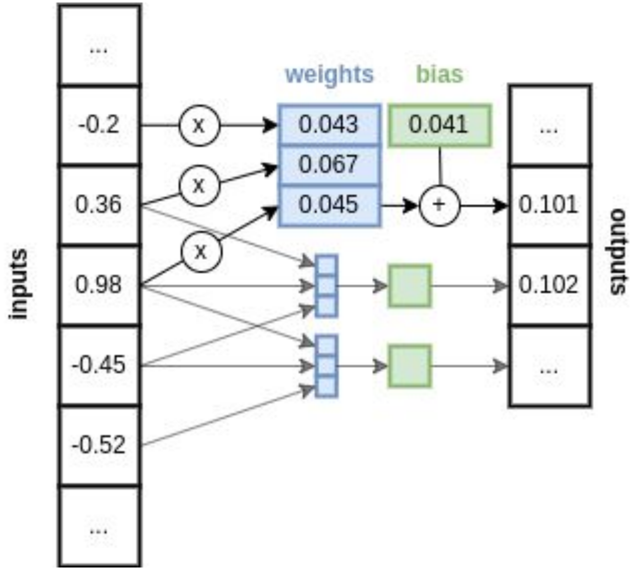
- So OK → Alexnet used CNN due to the translation-invariance property
- The next revolution was ResNets from Microsoft, using CNNs with some "residual connections", allowing for easier gradient propagation leading to depth > 100s of layers
  - an output of layer 1 gets element-wise added to the output of layer 3, etc.
- The list goes on, with the latest members being Transformers (GPTs T) and Mamba (new hype)
- Major types of NNs: MLPs, CNNs, Recurrent NNs (RNNs) and their subclass LSTMs (basically RNN with some complex memory modeling), and Transformers
- Anything else (GANs, RBFs, autoencoders etc.) you can think of them as variations on these

Neural Network Types

- A deeper investigation of NNs is outside our scope, so we'll stick to the ones we can easily use
- MLPs and CNNs are extremely versatile, MLPs have proved their worth historically, and even very small CNNs can realize useful tasks:
  - <400 KB CNN, doing ES→EN translation: https://github.com/HyperbeeAI/nanotranslator
    Almost as good as Google Translate on news articles!
  - Keyword spotting ("up", "down", "yes", …):
    https://www.analog.com/en/resources/design-notes/keywords-spotting-using-the-max78000.html
  - …

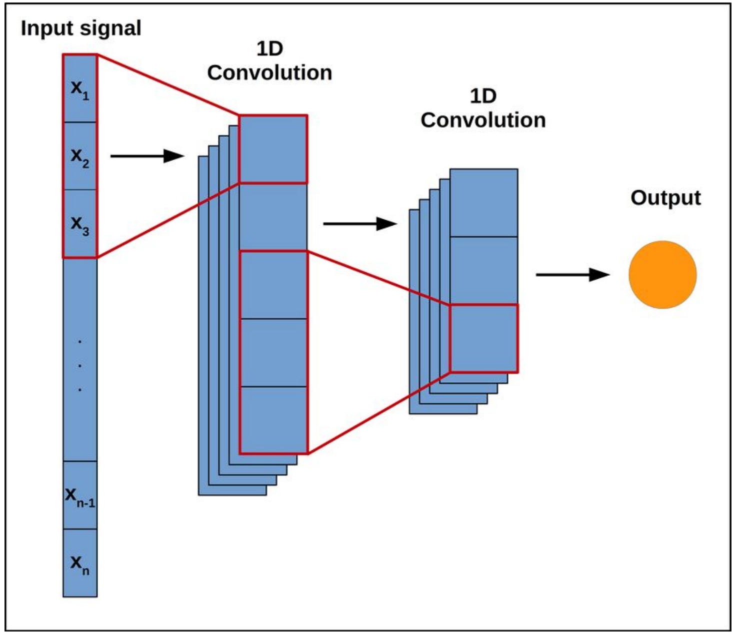
The CNN

- Let's break the CNN down then, just like we did with the MLP
- The neuron has a similar format, it does a linear transformation with weights and biases first, but this time there aren't separate weights for every input element
- There is an FIR filter weight set, this slides over the input elements (i.e., convolution), and a bias gets added on top
- Then, the output of the convolution is fed to a nonlinear activation function, typically a ReLU function (y: =x if x>0 else =0)
  - this used to be sigmoid or tanh in the older times to keep the network differentiable in the backward pass, but it was shown later that residual functions like the ReLU led to easier optimization landscapes, so people switched to those.



The CNN

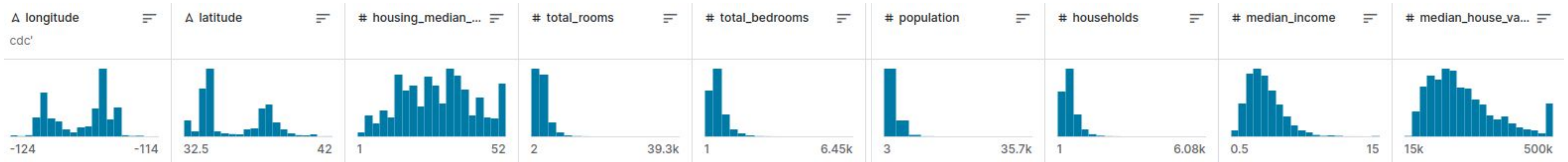
- CNNs get wider on the "channel" axis: multiple FIRs are applied in parallel on the same input
  - CNNs are "fully-connected" like the MLPs on this channel axis
- So, the equation of the CNN is simple too! It's just FIR filters instead of the matmul on MLP
- There are infinitely more complicated CNNs you can find out there, with different types of additional layers too, but for this course we will not dive deeper into that direction, we'll just try to run a simple NN on an FPGA



img src:
https://www.researchgate.net/publication/344229502_A_Novel_Deep_Learning_Model_for_the_Detection_and_Identification_of_Rolling_Element-Bearing_Faults/figures?lo=1

NN Inference on FPGA

- Let's go back to the MLP and start thinking about the FPGA implementation over an example
- Example: Predicting California housing prices
  - 1x8 input vector: median income, house age, avg. # of rooms, avg. # of bedrooms, total population, avg. # of household members, house latitude, house longitude
  - 1x1 output: median house price prediction
- Some are integers, some are floats, mixed

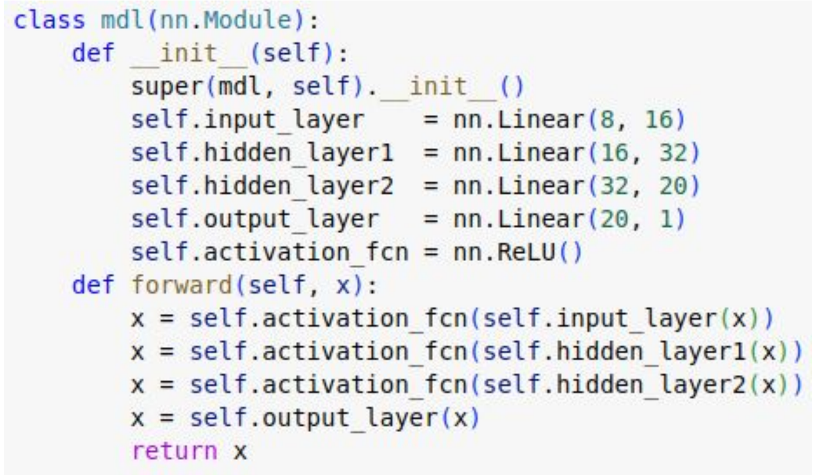


NN Inference on FPGA

- Assume that we'll use the following MLP →

- This basically has 2 hidden layers, 1 input and 1 output layer, and all layers other than the final output layer has ReLU activation functions.

- # of parameters on each layer:
  - Input layer → 8*16 weights, 16 biases
  - Hidden layer 1 → 16*32 weights, 32 biases
  - Hidden layer 2 → 32*20 weights, 20 biases
  - Output layer → 20*1 weights, 1 bias



```python
class mdl(nn.Module):
    def __init__(self):
        super(mdl, self).__init__()
        self.input_layer     = nn.Linear(8, 16)
        self.hidden_layer1   = nn.Linear(16, 32)
        self.hidden_layer2   = nn.Linear(32, 20)
        self.output_layer    = nn.Linear(20, 1)
        self.activation_fcn = nn.ReLU()
    def forward(self, x):
        x = self.activation_fcn(self.input_layer(x))
        x = self.activation_fcn(self.hidden_layer1(x))
        x = self.activation_fcn(self.hidden_layer2(x))
        x = self.output_layer(x)
        return x
```

- That's 1300 weights, and 69 biases, 1369 parameters in total

- Not huge, but compared to the 2-3 parameters we had on earlier labs, it's big

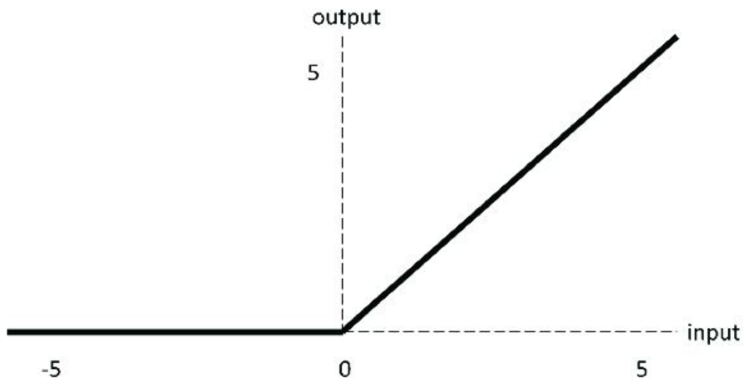
NN Inference on FPGA

- The equation for this MLP will not be as simple as the first one we saw with 1 hidden layer since this time the output of hidden layer 1 is fed to a hidden layer 2 instead of the output

- The types of operators inside do not change though. We will still have to do add and mult

- There's one new operator → ReLU

  - How can this be implemented on the FPGA efficiently, any guesses?



Img src: https://www.researchgate.net/publication/333411007_Multi-Classification_of_Brain_Tumor_Images_Using_Deep_Neural_Network/figures?lo=1&utm_source=google&utm_medium=organic

NN Inference on FPGA

- The equation for this MLP will not be as simple as the first one we saw with 1 hidden layer since this time the output of hidden layer 1 is fed to a hidden layer 2 instead of the output

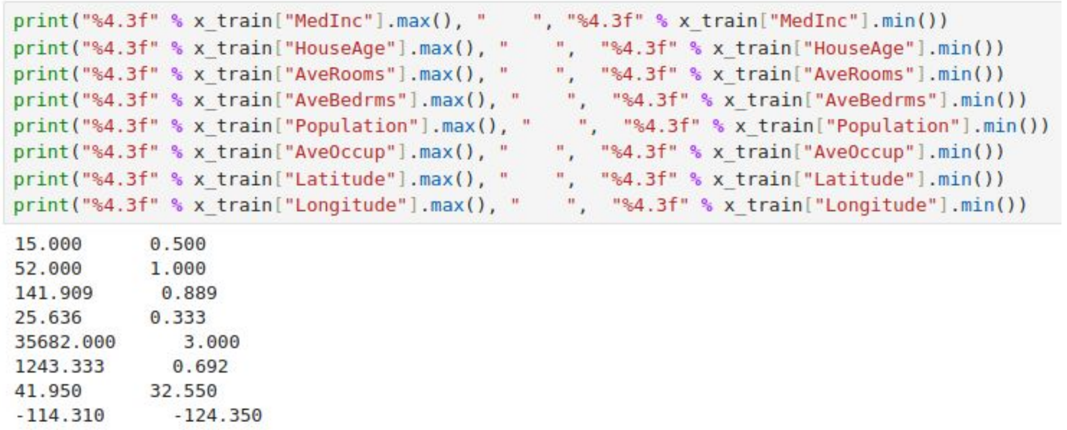
- The types of operators inside do not change though. We will still have to do add and mult

- There's one new operator → ReLU
  - Put an if on the sign bit, if it's 1, assign 0!

- For more complicated activation functions like the sigmoid, you usually need LUT-based implementations
  - example: https://github.com/dicearr/neuron-vhdl/blob/master/src/sigmoid.vhd

NN Inference on FPGA

- OK, the operators are covered, now for the values that go into those operators…
- Just like in the FIR example, we'll have to choose formats for every number and quantize
- Let's start with the inputs
- There's some engineering to be done, we need to check the min-max values and how the values are distributed to see what Q format should be assigned to each value



```
print("%4.3f" % x_train["MedInc"].max(), "     ", "%4.3f" % x_train["MedInc"].min())
print("%4.3f" % x_train["HouseAge"].max(), "     ", "%4.3f" % x_train["HouseAge"].min())
print("%4.3f" % x_train["AveRooms"].max(), "     ", "%4.3f" % x_train["AveRooms"].min())
print("%4.3f" % x_train["AveBedrms"].max(), "     ", "%4.3f" % x_train["AveBedrms"].min())
print("%4.3f" % x_train["Population"].max(), "     ", "%4.3f" % x_train["Population"].min())
print("%4.3f" % x_train["AveOccup"].max(), "     ", "%4.3f" % x_train["AveOccup"].min())
print("%4.3f" % x_train["Latitude"].max(), "     ", "%4.3f" % x_train["Latitude"].min())
print("%4.3f" % x_train["Longitude"].max(), "     ", "%4.3f" % x_train["Longitude"].min())
```

```
15.000       0.500
52.000       1.000
141.909       0.889
25.636       0.333
35682.000       3.000
1243.333       0.692
41.950       32.550
-114.310       -124.350
```

NN Inference on FPGA

- Next, we'll do the same for parameters, just like we did for FIR filter coefficients
- Afterwards, we'll analyze the computation for the Q formats we chose and see how that affects our prediction accuracy
  - The quantization error didn't hurt us too much in the FIR case, but it will hurt us now
  - The consequent layers will amplify the quantization error, possibly creating even gibberish predictions compared to our starting point
- We'll iterate, and try to see if we can get better accuracy with more bits (i.e., less quantization error). Once we're OK with the accuracy we see with simulated FPGA implementation, we'll then transfer this network to VHDL, implement it, and do the simulation in VHDL, just like we did in the FIR case.

next → california housing example

🙋🏽‍♂️