Lecture 12. Feed-forward Neural Networks and Backpropagation

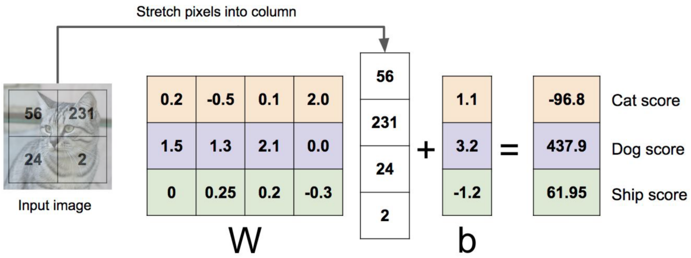
Bao Wang
Department of Mathematics
Scientific Computing and Imaging Institute
University of Utah
Math 5750/6880, Fall 2023

Multi-class Logistic Regression

Linear classifier

- $$f(\boldsymbol{x}, \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}.$$

- Consider an image classification task with 3 classes (cat/dot/frog), where each image has 4 pixels. In this case, $\boldsymbol{W} \in \mathbb{R}^{3\times 4}$ and $\boldsymbol{b} \in \mathbb{R}^3$.



- Is the output good? How to find optimal $\boldsymbol{W}$ and $\boldsymbol{b}$?

How to find $\boldsymbol{W}$ and $\boldsymbol{b}$?

- Define a **loss function** that quantifies our unhappiness with the scores across the training data.

- **Optimization:** Find the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ that minimize the loss function. Note that we can absorb $\boldsymbol{b}$ into $\boldsymbol{W}$ by introduce an additional dimension (all 1s) to the feature.

Loss function

A loss function tells how good our current classifier is.

- Given a dataset of examples $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$, where $\boldsymbol{x}_i$ is image and y_i is (integer) label.

- Loss over the dataset is the average of loss over examples

$$L = \frac{1}{N} \sum_i L_i\Big(f(\boldsymbol{x}_i, \boldsymbol{W}), y_i\Big).$$

Regularization

$$L(\boldsymbol{W}) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i\Big(f(\boldsymbol{x}_i, \boldsymbol{W}), y_i\Big)}_{\text{data loss}} + \underbrace{\lambda R(\boldsymbol{W})}_{\text{regularization}} ,$$

where λ is the regularization strength.

- L_2 regularization: $R(\boldsymbol{W}) = \sum_k \sum_l W_{k,l}^2$.
- L_1 regularization: $R(\boldsymbol{W}) = \sum_k \sum_l |W_{k,l}|$.
- Elastic net: $R(\boldsymbol{W}) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$.

Softmax classifier – Multinomial logistic regression

- Interpret raw classifier scores as probabilities.

- Given

$$\boldsymbol{s} = f(\boldsymbol{x}_i; \boldsymbol{W}),$$

how to convert $\boldsymbol{s}$ into probabilities?

Softmax classifier – Multinomial logistic regression

- Interpret raw classifier scores as probabilities.

$$\boldsymbol{s} = f(\boldsymbol{x}_i; \boldsymbol{W}) \Rightarrow P(Y = k | X = \boldsymbol{x}_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \quad \text{(softmax function)}.$$

- We need to maximize probability of correct class, i.e., minimize

$$L_i = -\log P(Y = y_i | X = x_i),$$

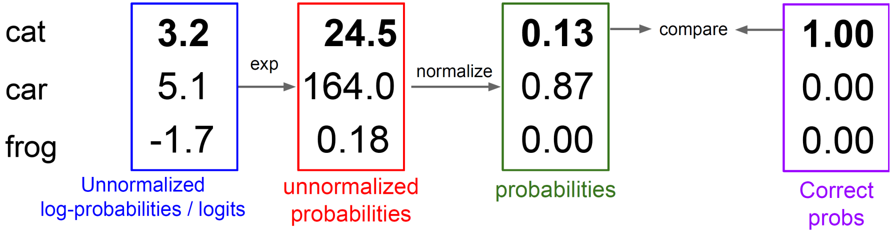
and therefore the loss contributed by $\boldsymbol{x}_i$ is

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right).$$

- When L_i will be minimized?

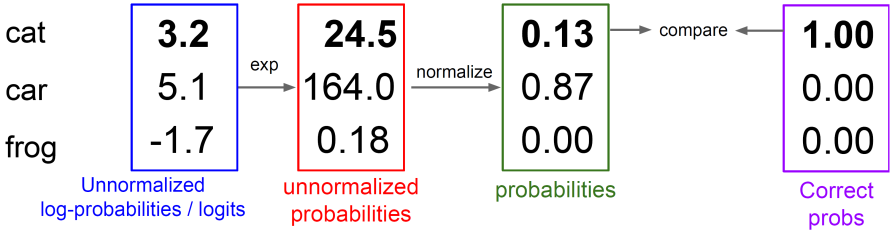
- When L_i will be minimized?

$$e^{s_{y_i}} = \sum_j e^{s_j}.$$



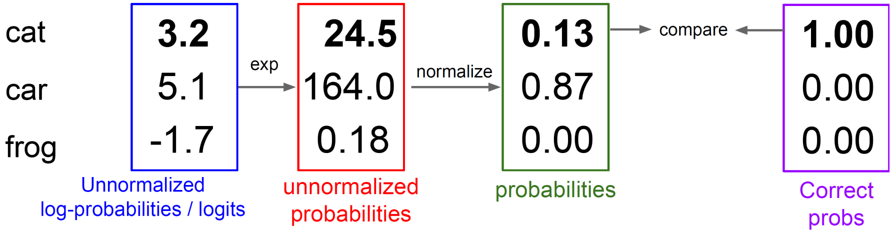
$$L_i = -\log P(Y = y_i | X = x_i) = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) = -\log(0.13) = 2.04.$$

Kullback-Leibier (KL) divergence



$$D_{KL}(P||Q) = \sum_y P(y) \log \frac{P(y)}{Q(y)}.$$

Cross entropy



$$H(P, Q) = H(P) + D_{KL}(P||Q),$$

where $H(P) = -\sum_y P(y) \log P(y)$ is the entropy of the distribution P.

Gradient descent (GD)

- So far, we have the following loss function

$$L(\boldsymbol{W}) = \frac{1}{N}\sum_{i=1}^{N} L_i(\boldsymbol{x}_i, y_i, \boldsymbol{W}) + \lambda R(\boldsymbol{W}),$$

and the gradient is

$$\nabla_{\boldsymbol{W}} L(\boldsymbol{W}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{W}} L_i(\boldsymbol{x}_i, y_i, \boldsymbol{W}) + \lambda \nabla_W R(\boldsymbol{W}),$$

then perform the following gradient descent

$$\boldsymbol{W}^{k+1} = \boldsymbol{W}^k - s\nabla_{\boldsymbol{W}} L(\boldsymbol{W}^k).$$

- Compute $\nabla_{\boldsymbol{W}} L(\boldsymbol{W})$ can be very expensive when N is large.

Stochastic gradient descent (SGD)

- Replace the true gradient above with the following mini-batch gradient

$$\frac{1}{n}\sum_{j=1}^{n} \nabla_{\boldsymbol{W}} L_{i_j}(\boldsymbol{x}_{i_j}, v_{i_j}, \boldsymbol{W}) + \lambda \nabla_{\boldsymbol{W}} R(\boldsymbol{W}), \quad n \ll N.$$

Neural Networks

Linear function

- Linear function: $f(\boldsymbol{x}; \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{x}$.

Neural network

- Linear function: $f(\boldsymbol{x}; \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{x}$.
- 2-layer neural network

$$f(\boldsymbol{x}; \boldsymbol{W}_1, \boldsymbol{W}_2) = \boldsymbol{W}_2 \max(0, \boldsymbol{W}_1 \boldsymbol{x}),$$

where $\boldsymbol{x} \in \mathbb{R}^d$, $\boldsymbol{W}_q \in \mathbb{R}^{H \times d}$, and $\boldsymbol{W}_2 \in \mathbb{R}^{C \times H}$.

$\max(0, *)$ is the ReLU activation.

Neural network

- Linear function: $f(\boldsymbol{x}; \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{x}$.

- 2-layer neural network

$$f(\boldsymbol{x}; \boldsymbol{W}_1, \boldsymbol{W}_2) = \boldsymbol{W}_2 \max(0, \boldsymbol{W}_1 \boldsymbol{x}),$$

where $\boldsymbol{x} \in \mathbb{R}^d$, $\boldsymbol{W}_q \in \mathbb{R}^{H \times d}$, and $\boldsymbol{W}_2 \in \mathbb{R}^{C \times H}$.

- 3-layer neural network

$$f(\boldsymbol{x}; \boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{W}_3) = \boldsymbol{W}_3 \max(0, \boldsymbol{W}_2 \max(0, \boldsymbol{W}_1 \boldsymbol{x})),$$

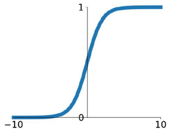
where $\boldsymbol{x} \in \mathbb{R}^d$, $\boldsymbol{W}_1 \in \mathbb{R}^{H_1 \times d}$, $\boldsymbol{W}_2 \in \mathbb{R}^{H_2 \times H_1}$, and $\boldsymbol{W}_3 \in \mathbb{R}^{C \times H_2}$.

$\max(0, *)$ is the ReLU activation.

Activation functions

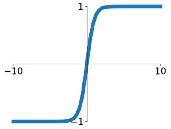
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



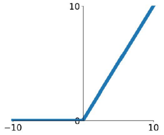
tanh

$\tanh(x)$



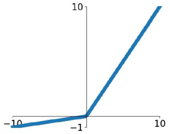
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

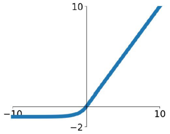


Maxout

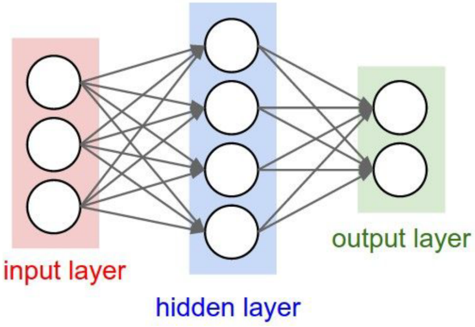
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

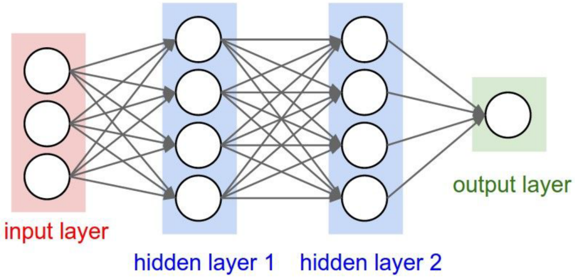
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Architectures



2-layer



3-layer

Backpropagation

A simple example

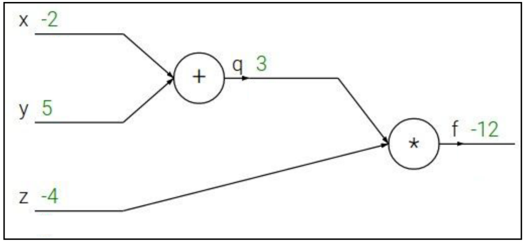
- Consider the following function

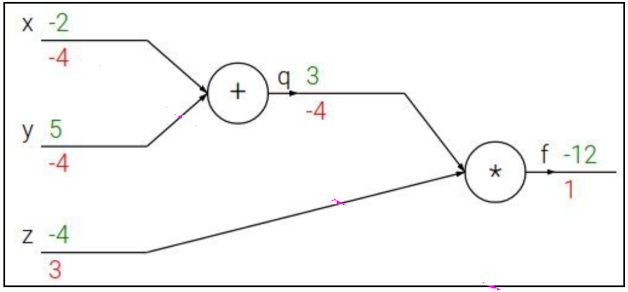
$$f(x,y,z) = (x+y)z,$$

e.g., $x=-2, y=5, z=-4$. Note that $f(x,y,z)$ can be decomposed as follows

$$q = x+y, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1; \quad f = qz, \frac{\partial f}{\partial q} = z, \ \frac{\partial f}{\partial z} = q.$$

- We need to compute $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$.
- Note the above function can be represented as the following computational graph





Note that

$$\frac{\partial f}{\partial f} = 1,$$

then

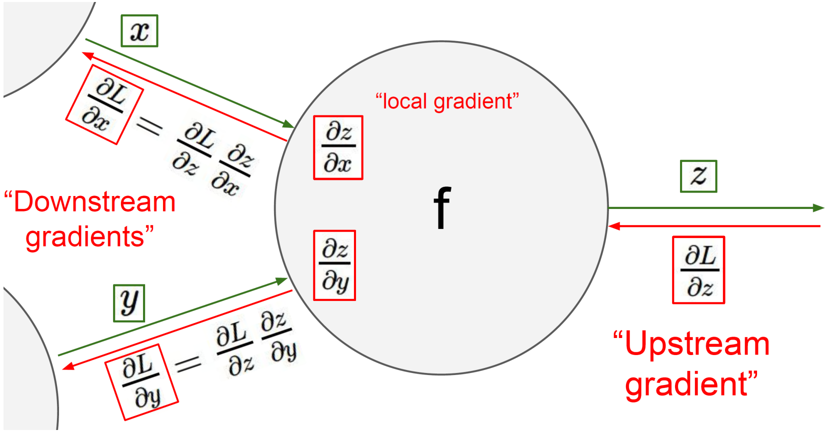
$$\frac{\partial f}{\partial z} = q = x + y = 3, \quad \frac{\partial f}{\partial q} = z = -4,$$

thus

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = -4 * 1 = -4; \quad \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = -4 * 1 = -4.$$

Chain rule

$$\underbrace{\frac{\partial f}{\partial x}}_{\text{Downstream gradient}} = \underbrace{\frac{\partial f}{\partial q}}_{\text{Upstream gradient}} \times \underbrace{\frac{\partial q}{\partial x}}_{\text{Local gradient}} .$$

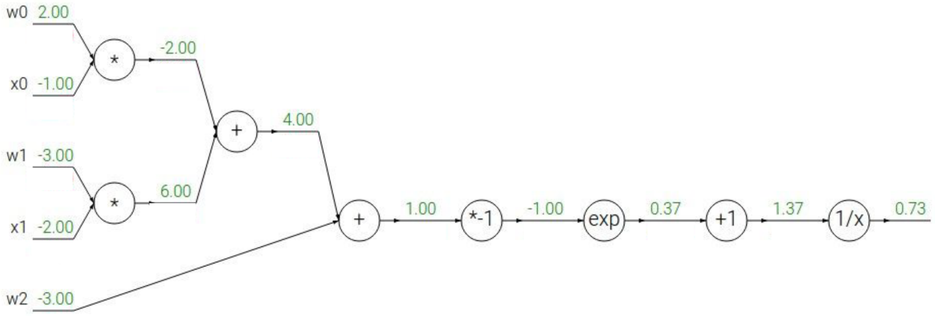


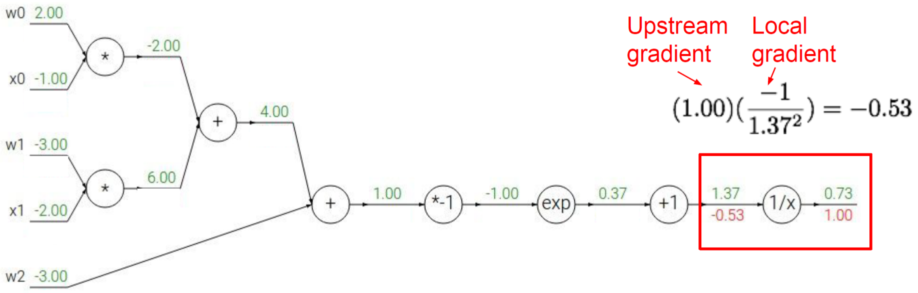
x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
z
"Downstream gradients"
$\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
"Upstream gradient"

Another example

$$f(\boldsymbol{w}, \boldsymbol{x}) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_2 + w_2)}},$$

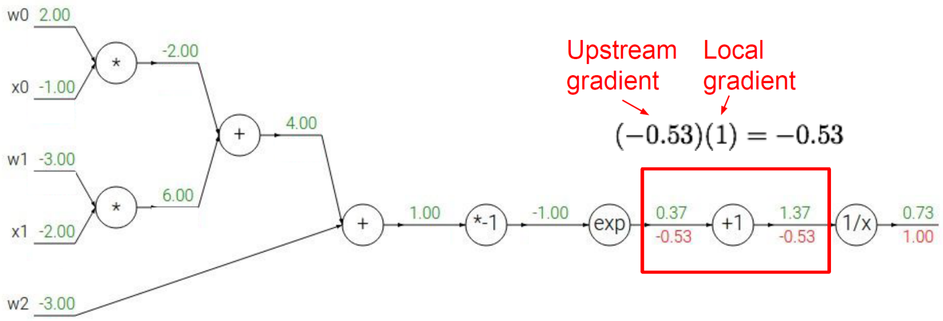
where $w_0 = 2, w_1 = -3, w_2 = -3, x_0 = -1,$ and $x_1 = -2$. The function can be represented by the following computational graph





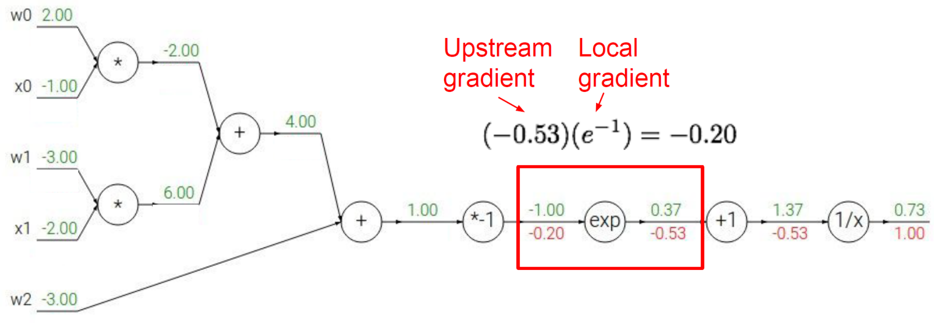
$$(1.00)\left(\frac{-1}{1.37^2}\right) = -0.53$$

$$f(x) = \frac{1}{x} \Rightarrow \frac{df}{dx} = -\frac{1}{x^2}.$$



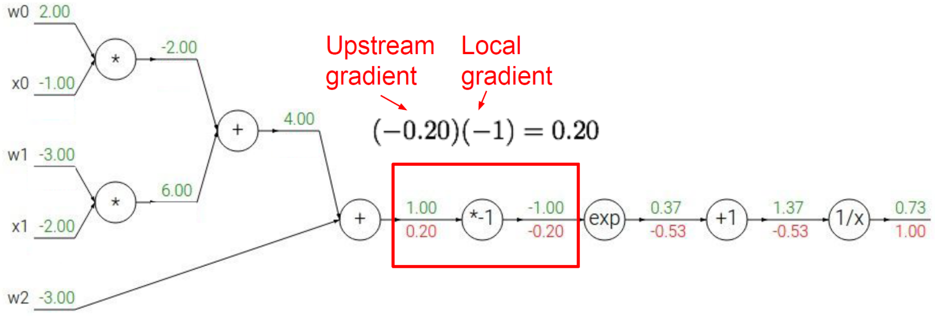
$$(-0.53)(1) = -0.53$$

$$f_c(x) = c + x \Rightarrow \frac{df_c}{dx} = 1.$$



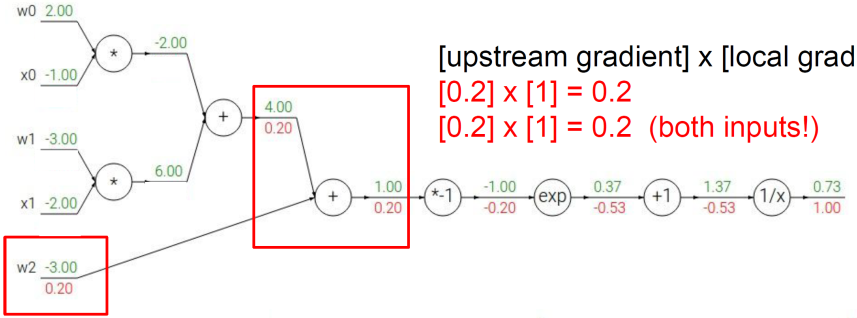
$$(-0.53)(e^{-1}) = -0.20$$

$$f(x) = e^x \Rightarrow \frac{df}{dx} = e^x.$$



$$(-0.20)(-1) = 0.20$$

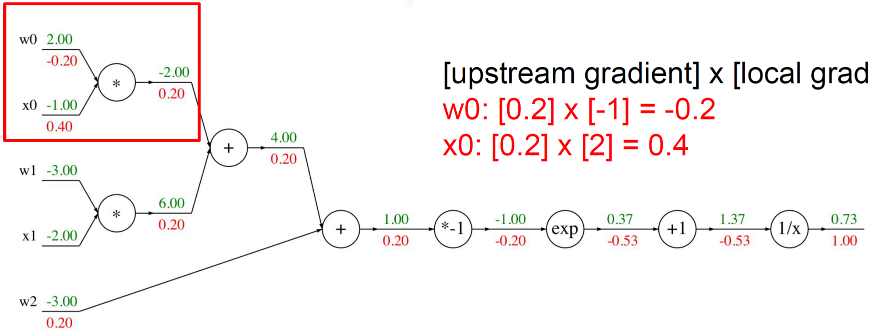
$$f_a(x) = ax \Rightarrow \frac{df}{dx} = a.$$



[upstream gradient] x [local gradient]

[0.2] x [1] = 0.2

[0.2] x [1] = 0.2 (both inputs!)



[upstream gradient] x [local gradient]

w0: [0.2] x [-1] = -0.2

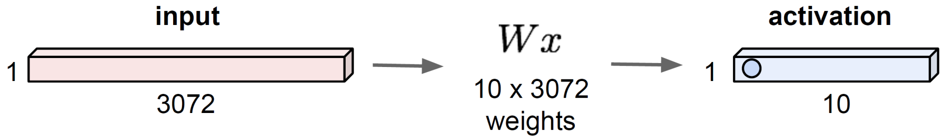
x0: [0.2] x [2] = 0.4

Convolutional Neural Networks

How linear model classify images?

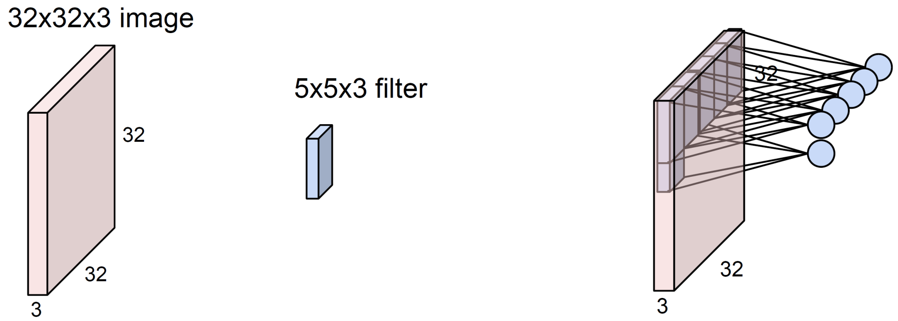
Image: height $\times$ width $\times$ channels, e.g. $32 \times 32 \times 3$.

- First stretch the image to a vector $\boldsymbol{x} \in \mathbb{R}^{3072 \times 1}$.
- Feed $\boldsymbol{x}$ into the model $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x}$ (fully-connected layer).



Convolution layer

Preserve the spatial structure of the input image.



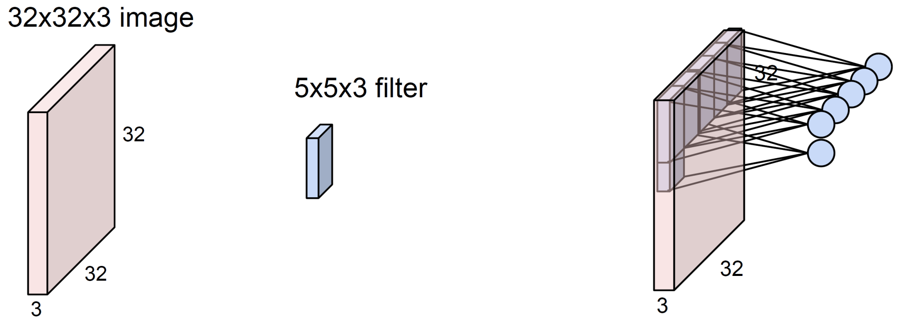
Convolve the filter with the image, i.e. "slide over the image spatially over height and width/or over height, width, and channel, computing dot products".

Recall convolution of two functions

$$(f * g)(y) = \int_{-\infty}^{\infty} f(x) \cdot g(y - x) dx.$$

Convolution layer

Preserve the spatial structure of the input image.

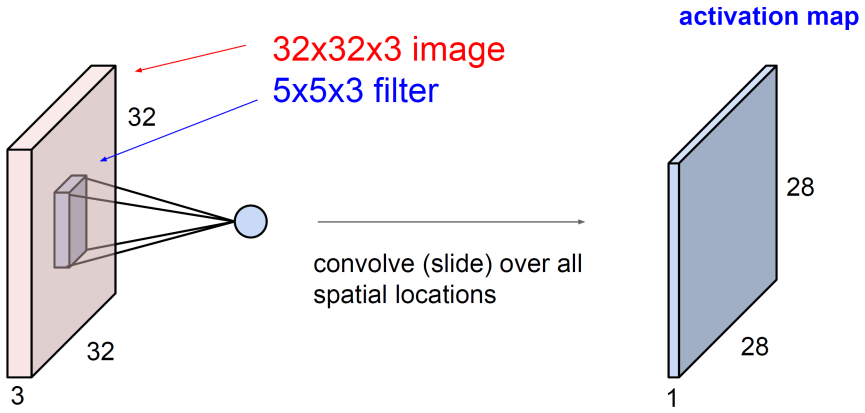


Convolve the filter with the image, i.e. "slide over the image spatially over height and width/or over height, width, and channel, computing dot products".

Recall convolution of two signals

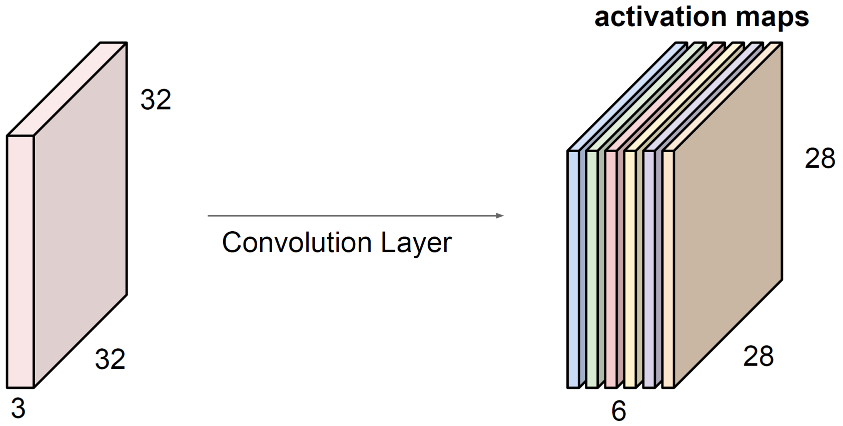
$$f(x, y) * g(x, y) = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f(n_1, n_2) \cdot g(x - n_1, y - n_2).$$

Activation map



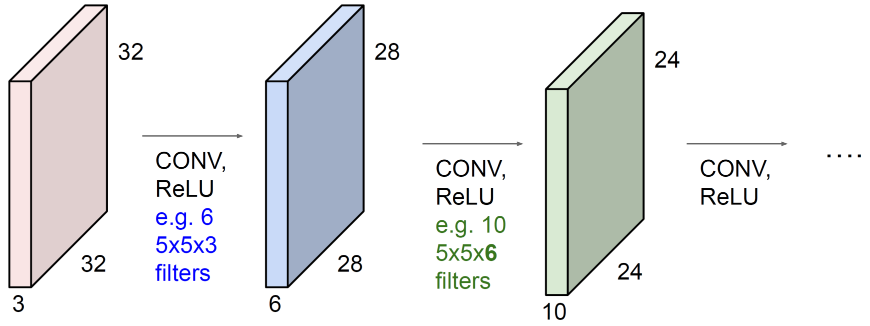
Multiple activation maps

We can use multiple filters to get multiple activation maps.

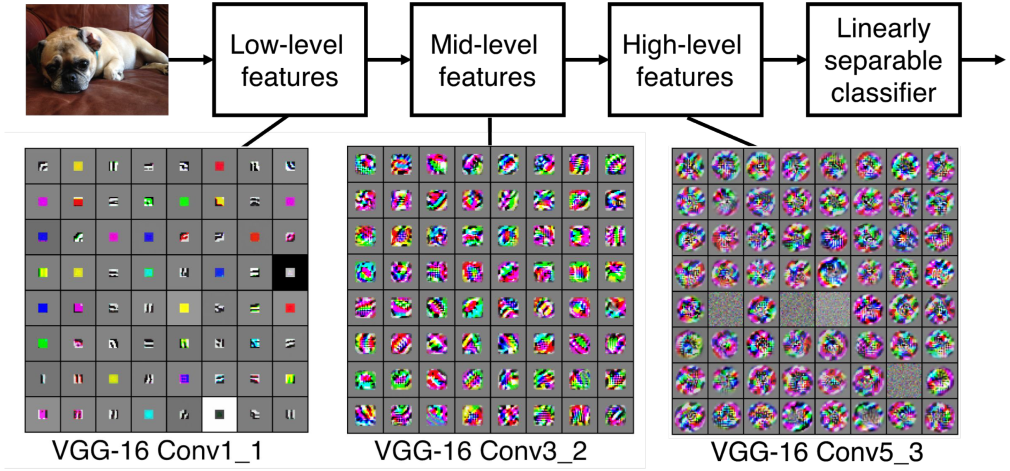


Convolutional neural networks (CNNs)

CNN is a sequence of convolution layers, interspersed with activation functions.

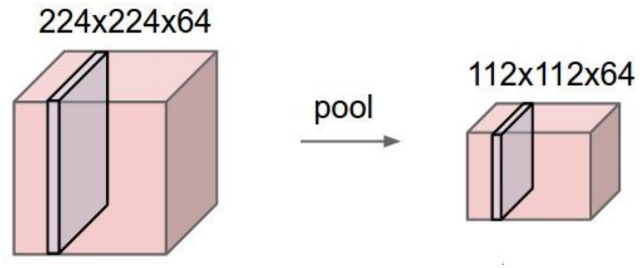


CNN for representation learning

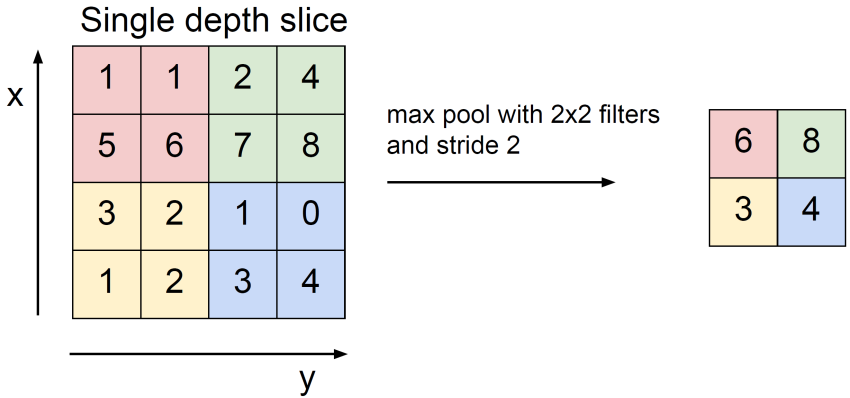


Pooling layer

Pooling layer is used to make the representations smaller and more manageable, which operates over each activation map independently.



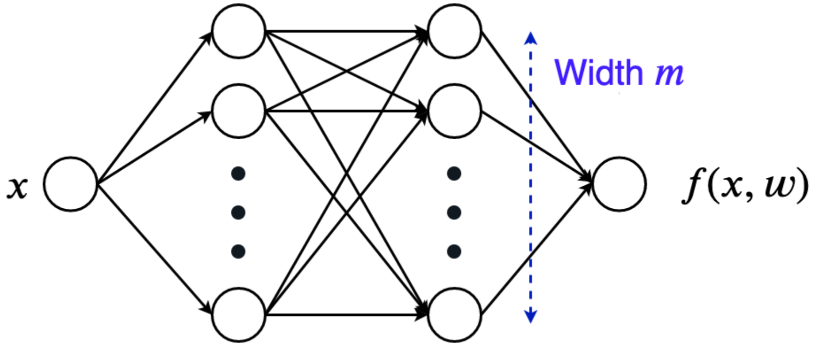
Max pooling


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

Neural Networks vs. Kernel Methods

- Is there any theoretical guarantee in training neural networks using gradient descent?

How does neural network relate to kernel method?

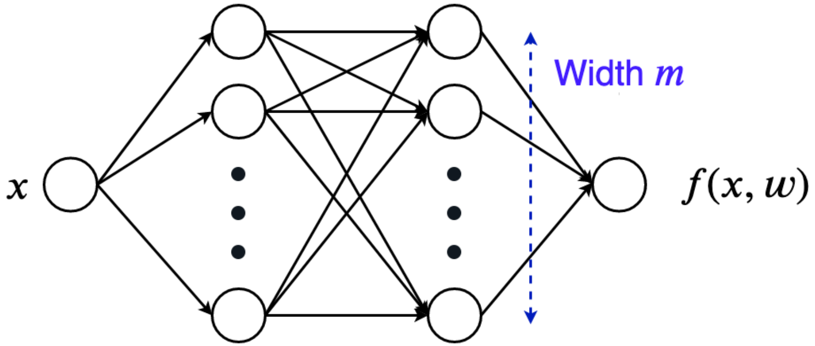


- Neural networks in the infinite-width regime simplify to linear models with a kernel called the neural tangent kernel.

- Under this regime, we can show that gradient descent will converge to 0 training loss.

A 1D example

• Let us start with a very simple example with 1D input and 1D output. In particular, a simple 2-hidden layer ReLU network with width m.



• Denote the neural network function as $f(x, \boldsymbol{w})$ where x is the input and $\boldsymbol{w}$ is the combined vector of weights (say of size p).
• Assume the training dataset is $\{\bar{x}_i, \bar{y}_i\}_{i=1}^{N}$.

Let us consider performing full-batch gradient descent on the following least squares loss

$$L(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\Big(f(\bar{x}_i, \boldsymbol{w}) - \bar{y}_i\Big)^2.$$

• Stack all the output dataset value $\bar{y}_i$ into a single vector of size N, and call it $\bar{\boldsymbol{y}}$.

• Stack all the model outputs for each input, $f(\bar{x}_i, \boldsymbol{w})$ into a single prediction vector $\boldsymbol{y}(\boldsymbol{w}) \in \mathbb{R}^N$, i.e., $\boldsymbol{y}(\boldsymbol{w})_i = f(\bar{x}_i, \boldsymbol{w})$.

• Our loss simplifies to

$$L(\boldsymbol{w}) = \frac{1}{N}\cdot\frac{1}{2}\|\boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}}\|_2^2.$$

We can rescale it and consider $L(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}}\|_2^2$.

How does the weight change under gradient descent?

- Consider the *relative change* in the norm of the weight vector from initialization:

$$\frac{\|\boldsymbol{w}(n) - \boldsymbol{w}_0\|_2}{\|\boldsymbol{w}_0\|_2}$$

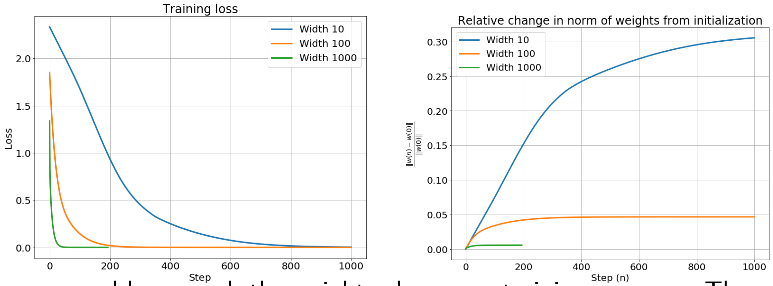


Figure: Loss curves and how much the weights change as training progress. The weight norms are calculated using all parameters in the model stacked into a single vector. Also, all other hyperparameters (like learning rate) are kept constant.

- The weights don't change much at all for larger hidden widths. – What can we do to analyze $f(x, \boldsymbol{w})$?

Linearize neural networks

- Taylor expand the network function **w.r.t. the weights** around its **initialization**. (Note in training, the weights do not change much.)

$$f(x, \boldsymbol{w}) \approx f(x, \boldsymbol{w}_0) + \nabla_{\boldsymbol{w}} f(x, \boldsymbol{w}_0)^\top (\boldsymbol{w} - \boldsymbol{w}_0).$$

- We've turned the non-linear neural network function into a linear function of the weights.

- Using our more concise vector notation for the model outputs on a specific dataset we can rewrite as:

$$\boldsymbol{y}(\boldsymbol{w}) \approx \boldsymbol{y}(\boldsymbol{w}_0) + \nabla_{\boldsymbol{w}} \boldsymbol{y}(\boldsymbol{w}_0)^\top (\boldsymbol{w} - \boldsymbol{w}_0),$$

where $\boldsymbol{y}(\boldsymbol{w}) \in \mathbb{R}^{n \times 1}$, $\nabla_{\boldsymbol{w}} \boldsymbol{y}(\boldsymbol{w}_0)^\top \in \mathbb{R}^{n \times p}$, and $\boldsymbol{w} \in \mathbb{R}^{p \times 1}$ with n and p are the number of datapoints and parameters, respectively.

• Note that the initial output $\boldsymbol{y}(\boldsymbol{w}_0)$ and the model Jacobian $\nabla_{\boldsymbol{w}}\boldsymbol{y}(\boldsymbol{w}_0)$ are just constants. Thus

$$\boldsymbol{y}(\boldsymbol{w}) \approx \boldsymbol{y}(\boldsymbol{w}_0) + \nabla_{\boldsymbol{w}}\boldsymbol{y}(\boldsymbol{w}_0)^\top(\boldsymbol{w} - \boldsymbol{w}_0),$$

is just a **linear model in the weights**, so minimizing the least squares loss reduces to just doing **linear regression**!

• But, notice that the model function is still **non-linear in the input**. In fact, $\boldsymbol{y}(\boldsymbol{w}_0) + \nabla_{\boldsymbol{w}}\boldsymbol{y}(\boldsymbol{w}_0)^\top(\boldsymbol{w} - \boldsymbol{w}_0)$ is just a linear model using a **feature map** $\phi(\boldsymbol{x})$ which is the gradient vector at initialization:

$$\phi(x) = \nabla_{\boldsymbol{w}} f(x, \boldsymbol{w}_0).$$

Gradient descent dynamics

- We've considered the linearization of neural networks, $\boldsymbol{y}(\boldsymbol{w}_0) + \phi(x)(\boldsymbol{w} - \boldsymbol{w}_0)$.
- Let us now get into their training dynamics under gradient descent

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \eta \nabla_{\boldsymbol{w}} L(\boldsymbol{w}_k).$$

i.e.,

$$\frac{\boldsymbol{w}_{k+1} - \boldsymbol{w}_k}{\eta} = -\nabla_{\boldsymbol{w}} L(\boldsymbol{w}_k).$$

- Let $\eta \to 0$, we have

$$\frac{d\boldsymbol{w}(t)}{dt} = -\nabla_{\boldsymbol{w}} L(\boldsymbol{w}(t)).$$

This is called a **gradient flow**. To simplify notation, we denote time derivatives with a dot, the gradient flow is:

$$\dot{\boldsymbol{w}}(t) = -\nabla L(\boldsymbol{w}(t)).$$

Gradient flow

- Dropping the time variable, and substituting for the loss and taking the gradient, we get (note $L(\boldsymbol{w}(t)) = \frac{1}{2}\|\boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}}\|_2^2$.)

$$\dot{\boldsymbol{w}} = -\nabla \boldsymbol{y}(\boldsymbol{w})(\boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}}). \quad (\nabla \boldsymbol{y}(\boldsymbol{w}) = \nabla_{\boldsymbol{w}} \boldsymbol{y}(\boldsymbol{w}))$$

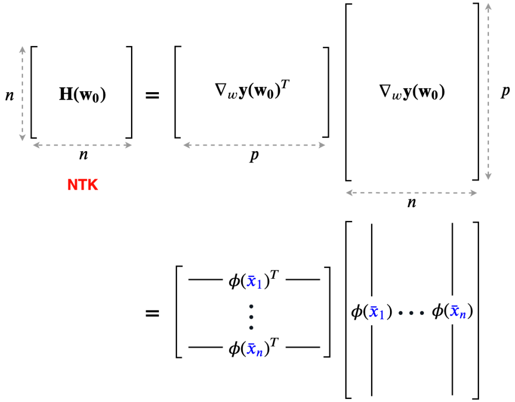
- We can now derive the dynamics of the model outputs $\boldsymbol{y}(\boldsymbol{w})$ (this is basically the **dynamics in function space**) induced by this gradient flow using the chain rule:

$$\dot{\boldsymbol{y}}(\boldsymbol{w}) = \nabla \boldsymbol{y}(\boldsymbol{w})^\top \dot{\boldsymbol{w}} = -\nabla \boldsymbol{y}(\boldsymbol{w})^\top \nabla \boldsymbol{y}(\boldsymbol{w}) \Big(\boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}} \Big).$$

- The quantity $\boldsymbol{H}(\boldsymbol{w}) := \nabla \boldsymbol{y}(\boldsymbol{w})^\top \nabla \boldsymbol{y}(\boldsymbol{w})$ is called the **neural tangent kernel** (NTK).

Neural tangent kernel

- Recall that the linearized model has a feature map $\phi(x) = \nabla_{\boldsymbol{w}} f(x, \boldsymbol{w}_0)$. The kernel matrix corresponding to this feature map is obtained by taking pairwise inner products between the feature maps of all the data points. This is exactly $\boldsymbol{H}(\boldsymbol{w}_0)$.



- $\boldsymbol{H}(\boldsymbol{w}_0)$ is $\sim$ positive definite if $p \gg n$.

Gradient flow

• If the model is close to its linear approximation, the Jacobian of the model outputs **does not change as training progresses**, i.e.,

$$\nabla \boldsymbol{y}(\boldsymbol{w}(t)) \approx \nabla \boldsymbol{y}(\boldsymbol{w}_0) \Rightarrow \boldsymbol{H}(\boldsymbol{w}(t)) \approx \boldsymbol{H}(\boldsymbol{w}_0).$$

• This is referred to as the **kernel regime**, because the tangent kernel stays constant during training. The training dynamics now reduces to a very simple **linear ordinary differential equation:**

$$\dot{\boldsymbol{y}}(\boldsymbol{w}) = -\boldsymbol{H}(\boldsymbol{w}_0)(\boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}}).$$

Gradient flow

- Clearly, $\boldsymbol{y}(\boldsymbol{w}) = \bar{\boldsymbol{y}}$ is an equilibrium of this ODE, and it corresponds to a train loss of 0.
- Let $\boldsymbol{u} = \boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}}$, the flow simplifies to

$$\dot{\boldsymbol{u}} = -\boldsymbol{H}(\boldsymbol{w}_0)\boldsymbol{u}.$$

- The solution of this ODE is given by a matrix exponential

$$\boldsymbol{u}(t) = \boldsymbol{u}(0)e^{-\boldsymbol{H}(\boldsymbol{w}_0)t}.$$

When over-parameterized ($p \gg n$), the NTK $\nabla\boldsymbol{y}(\boldsymbol{w}_0)^\top\nabla\boldsymbol{y}(\boldsymbol{w}_0)$ is positive definite.

•
$$\boldsymbol{u}(t) = \boldsymbol{u}(0)e^{-\boldsymbol{H}t},$$
where $\boldsymbol{H} := \boldsymbol{H}(\boldsymbol{w}_0)$ is positive definite.

• Let $\boldsymbol{H} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{U}^{-1}$ be the spectral decomposition of $\boldsymbol{H}$, where $\boldsymbol{U}$ is orthogonal and $\boldsymbol{D}$ is diagonal with diagonal entries be positive. Therefore, using the fact that

$$e^{-\boldsymbol{U}\boldsymbol{D}\boldsymbol{U}^{-1}t} = \sum_{k=0}^{\infty} \frac{1}{k!}\Big(\boldsymbol{U}(-\boldsymbol{D}t)\boldsymbol{U}^{-1}\Big)^k = \boldsymbol{U}\Big(\sum_{k=0}^{\infty} \frac{1}{k!}(-\boldsymbol{D}t)^k\Big)\boldsymbol{U}^{-1} = \boldsymbol{U}e^{-\boldsymbol{D}t}\boldsymbol{U}^{-1},$$

we have
$$\boldsymbol{u}(t) = \boldsymbol{u}(0)\boldsymbol{U}e^{-\boldsymbol{D}t}\boldsymbol{U}^{-1} \to 0,$$
where $(e^{-Dt})_{ii} = e^{-d_{ii}t}$.

Universal Approximation Theorem

Capacity of the Perceptron

- What kind of function can a neural network represent? Let is start with the simplest neural network, the Perceptron, a NN with a single hidden layer having 1 hidden unit with an activation function σ. Both the input layer and the weights are a $1 \times n$ vector.

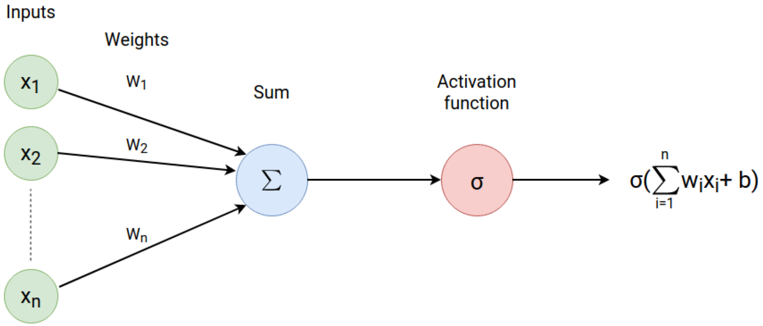


Figure: The perceptron.

- Perceptron:

$$\sigma\Big(\sum_{i=1}^{n} w_i x_i + b\Big) = \sigma\big(\boldsymbol{w}^\top \boldsymbol{x} + b\big),$$

where σ can be sigmoid: $\frac{1}{1+e^{-x}}$, tanh: $\frac{e^{2x}-1}{e^{2x}+1}$, ReLU: $\max(0, x)$ ...

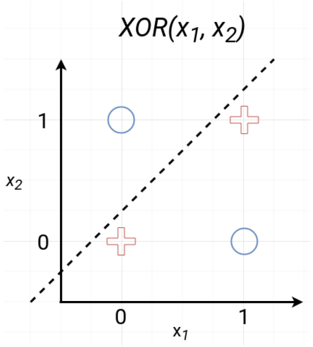


Figure: xOR is not linearly separable.

- Perceptron is a linear classifier, and as such it can't model an XOR.

Capacity of multiple neurons

- By allowing ourselves more than 1 neuron in the hidden layer, we can model a XOR and in fact, we get the simplest **universarial approximator**.

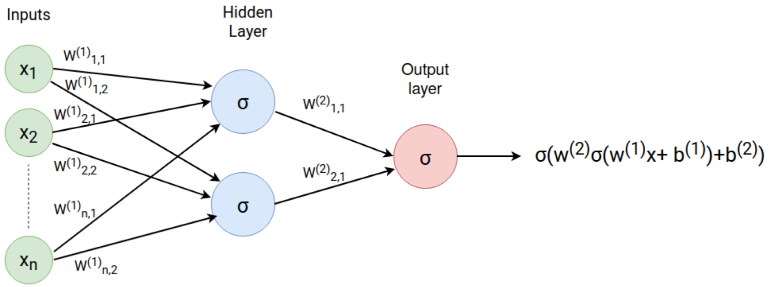


Figure: A NN with 2 hidden units.

Universal approximation theorem

- **Universal approximation theorem:** any continuous function $f : [0,1]^n \to [0,1]$ can be approximated arbitrarily well by a neural network with at least one hidden layer with a finite number of weights.

A visual proof is available at
`http://neuralnetworksanddeeplearning.com/chap4.html`.

Visual proof of universal approximation

Suppose we want to approximate a function with 1 input and 1 output like:

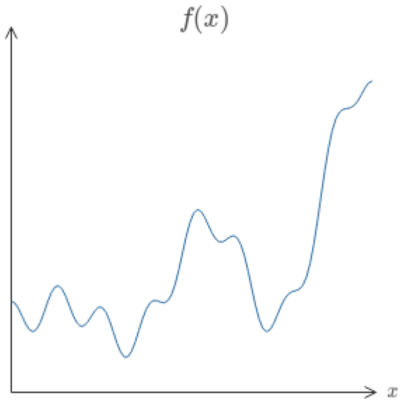


Figure: A continuous function.

We will first consider a simple NN with 2 hidden neurons that have a sigmoid activation function, and for now the output neuron will just be linear.

Visual proof of universal approximation – Step 1

Make a step function with 1 of the neuron.

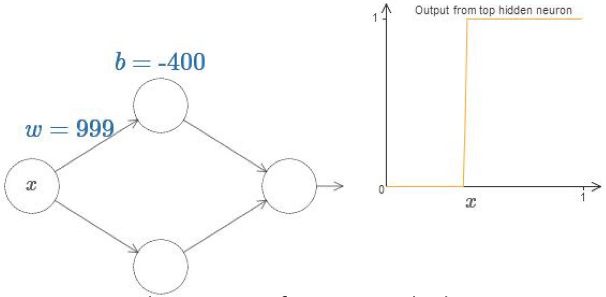


Figure: Making a step function with the top neuron.

Let us focus on the top hidden neuron first, by using a big weight on the top neuron we can approximate the step function with a sigmoid arbitrarily well, and by adjusting the bias we can place it anywhere. (The same argument can be made for the tanh activation, but not for ReLU.) Note that $\sigma(wx)$ becomes steeper as w increases.

Visual proof of universal approximation – Step 1

In the toy example above, we won't be interested in changing the weights of the first layer, they just have to be high enough, so we will just consider them to be constant, Additionally, to make the plots clearer, we will display the position of the step instead of the bias, which is easily computed with $s = -b/w$.
With these changes, the plot above becomes:

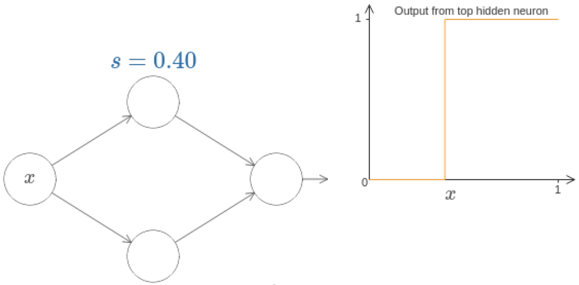


Figure: Making a step function with the top neuron.

Visual proof of universal approximation – Step 2

Make a “bin” with an opposite step function.

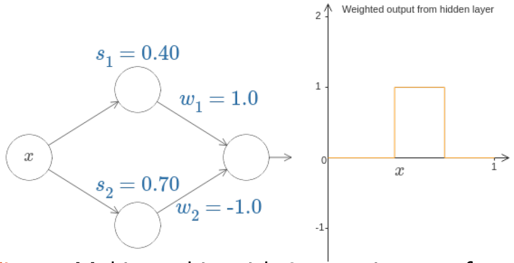


Figure: Making a bin with 2 opposite step functions.

As illustrated above, by using the other neuron to make a step function, and setting opposing weights in the second layer, we can effectively approximate a bin and control its position, size and height.

Visual proof of universal approximation – Step 2

Now you can probably see where this is going, to make things even clearer, we will just use 1 value for both w_1 and $-w_2$, called h, representing the height of the "bin".

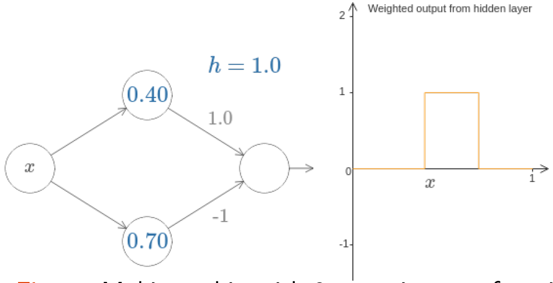
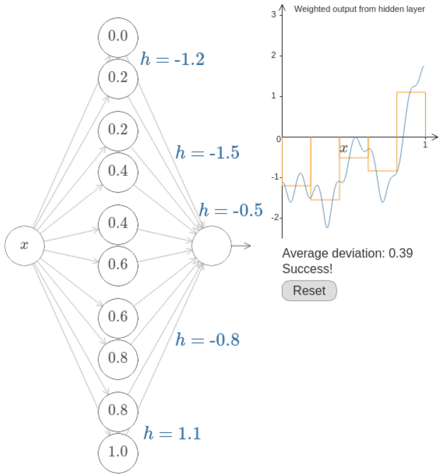


Figure: Making a bin with 2 opposite step functions.

Visual proof of universal approximation – Step 3

Discretize the function



Average deviation: 0.39
Success!

Figure: Approximating f with an histogram.

In this final step, we combine several “bias” to make an histogram approximating the function. Illustrated above is a very rough approximation using only 5 bins (10 hidden units), but we can obviously make it sharper by simply adding more bins.

Question: If we want to use this technique to approximate an L-Lipschitz function $f : \mathbb{R} \to \mathbb{R}$ on the interval $[0, 1]$ with an error at most ϵ at any point, how many bins would we need?

What if we don’t want linear neurons in the output layer? The above network’s output layer is linear, giving us the histogram approximating f, if we add a sigmoid activation function on the output we just have to approximate $\sigma^{-1} \circ f$ instead of f, which we can do with the same method.

Cybenko approximation by superposition of sigmoid function

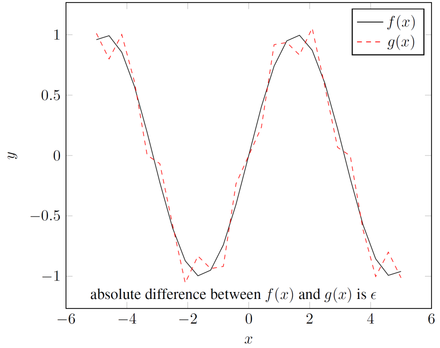
We first define a *sigmoidal function* σ as:

$$\sigma(x) \to \begin{cases} 1 & \text{as } x \to +\infty \\ 0 & \text{as } x \to -\infty \end{cases}$$

While sigmoidal functions are usually assumed to be monotonic increasing, this assumption is not necessary for this result.

Theorem. Let $C([0,1]^n)$ denote the set of all continuous function $[0,1]^n \to \mathbb{R}$, let σ be any sigmoid activation function then the finite sum of the form $f(\boldsymbol{x}) = \sum_{i=1}^{N} \alpha_i \sigma(\boldsymbol{w}_i^\top \boldsymbol{x} + b_i)$ is dense in $C([0,1]^n)$.

Informally, the above theorem says that for any $g \in C([0,1]^n)$ for any $\epsilon > 0$, there exists $f : \boldsymbol{x} \to \sum_{i=1}^{N} \alpha_i \sigma(\boldsymbol{w}_i^\top \boldsymbol{x} + b)$ such that $|f(\boldsymbol{x}) - g(\boldsymbol{x})| < \epsilon$ for all $\boldsymbol{x} \in [0,1]^n$.


$f(x)$
$g(x)$
absolute difference between $f(x)$ and $g(x)$ is ϵ
y
x