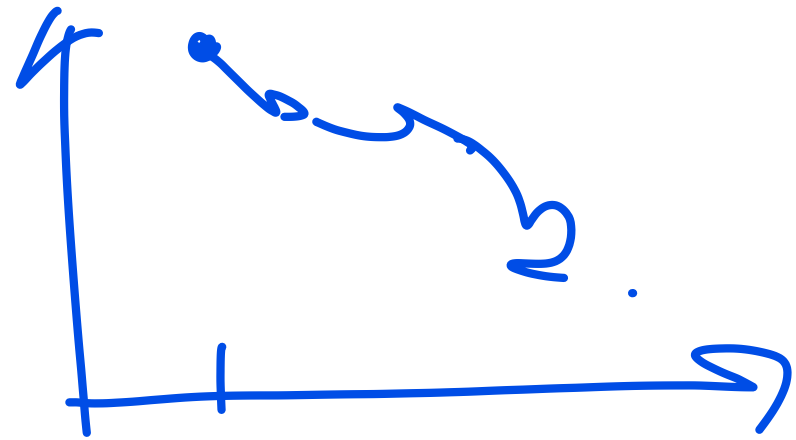
Optimizing Neural Networks

part 2 of 2

Initialization & activations

Part 2 of the optimization chunk: how to **initialize** the weights and which **activation** to use – then the neural-networks chapter wraps up.

Chapter 5 – Neural Networks



Deep Learning

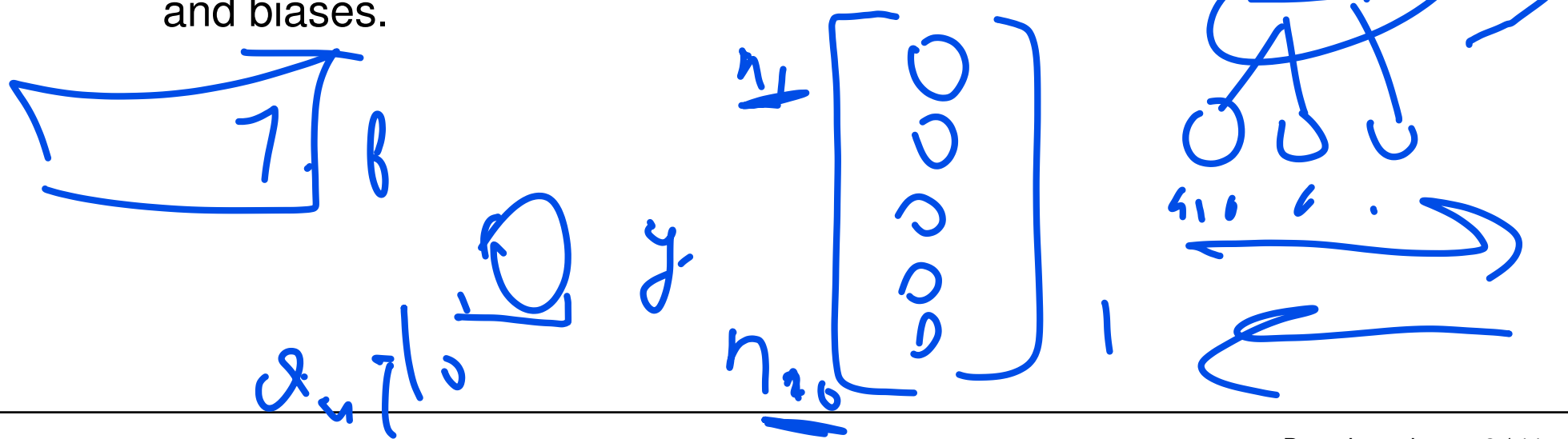
Network Initializations

Learning goals

- Why Initializaiton matters
- Weight Initializations
- Bias Initialization

PRACTICAL INITIALIZATION

- The weights (and biases) of a neural network must be assigned some initial values before training can begin.
- The choice of the initial weights (and biases) is crucial as it determines whether an optimization algorithm converges, how fast and whether to a point with high or low risk.
- Initialization strategies to achieve "nice" properties are difficult to find, because there is no good understanding which properties are preserved under which circumstances.
- In the following we seperate between the initialization of weights and biases.



WEIGHT INITIALIZATION

- It is important to initialize the weights randomly in order to "break symmetry". If two neurons (with the same activation function in a fully connected network) are connected to the same inputs and have the same initial weights, then both neurons will have the same gradient update in a given iteration and they will end up learning the same features.
- Furthermore, the initial weights should not be too large, because this might result in an explosion of weights or high sensitivity to changes in the input.
- Weights are typically drawn from a uniform distribution or a Gaussian centered at 0 with a small variance.
- Centering the initial weights around 0 can be seen as a form of regularization and it imposses that it is more likely that units do not interact with each other than they do interact.

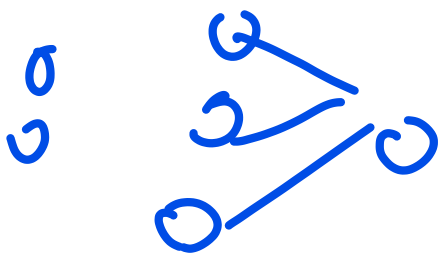
$\sim(0,1)$ MLE

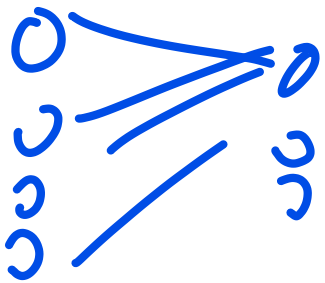
WEIGHT INITIALIZATION / 2

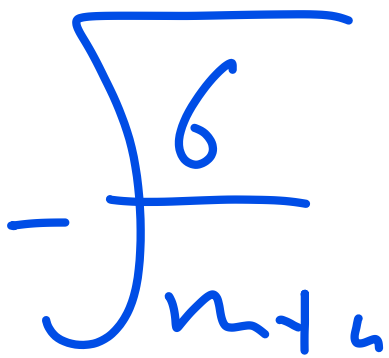
- Two common initialization strategies for weights are the 'Glorot initialization' and 'He initialization' which tune the variance of these distributions based on the topology of the network.
- **Glorot initialization** suggests to sample each weight of a fully connected layer with m inputs and n outputs from a uniform distribution

$$w_{j,k} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

The strategy is derived from the assumption that the network consists only of a chain of matrix multiplications with no nonlinearities.





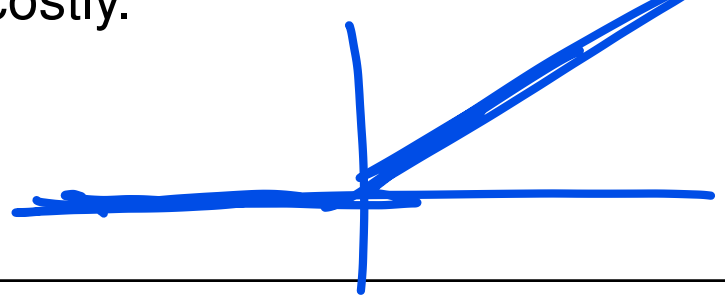


WEIGHT INITIALIZATION / 3

- **He initialization** is especially useful for neural networks with ReLU activations. Each weight of a fully connected layer with m inputs is sampled from a Gaussian distribution

$$w_{j,k} \sim N\left(0, \frac{2}{m}\right)$$

  The underlying derivation can be found in He et al. (2015).
- Since the initialization strategies of Glorot and He depend on the layer sizes, the initial weights for large layer sizes can become extremely small.
- Another strategy is to treat the weights as hyperparameters that can be optimized by hyperparameter search algorithms. This can be computationally costly.



WEIGHT INITIALIZATION: EXAMPLE

- We use a spiral planar data set to compare the following strategies: Zero initialization, random initialization (samples from $N\left(0, 1 \cdot 10^{-4}\right)$) and He initialization.
- For each strategy, a neural network with one hidden layer with 100 units, ReLU activation and Gradient Descent as optimizer was used.

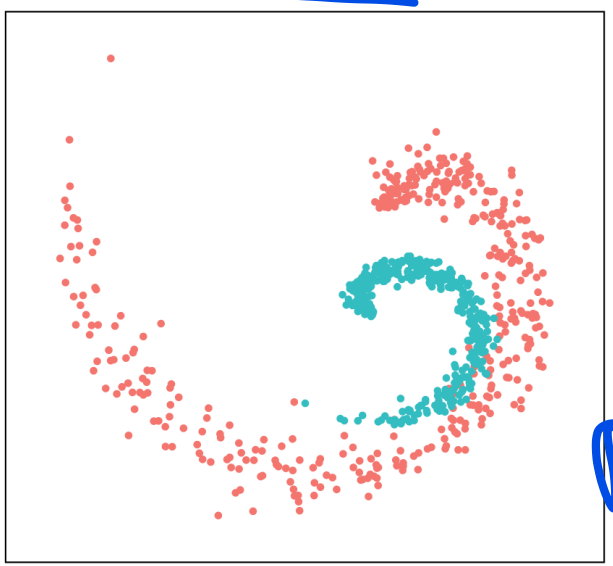


Figure: Simulated spiral planar data set with two classes (Ghatak, 2019).

WEIGHT INITIALIZATION: EXAMPLE / 2

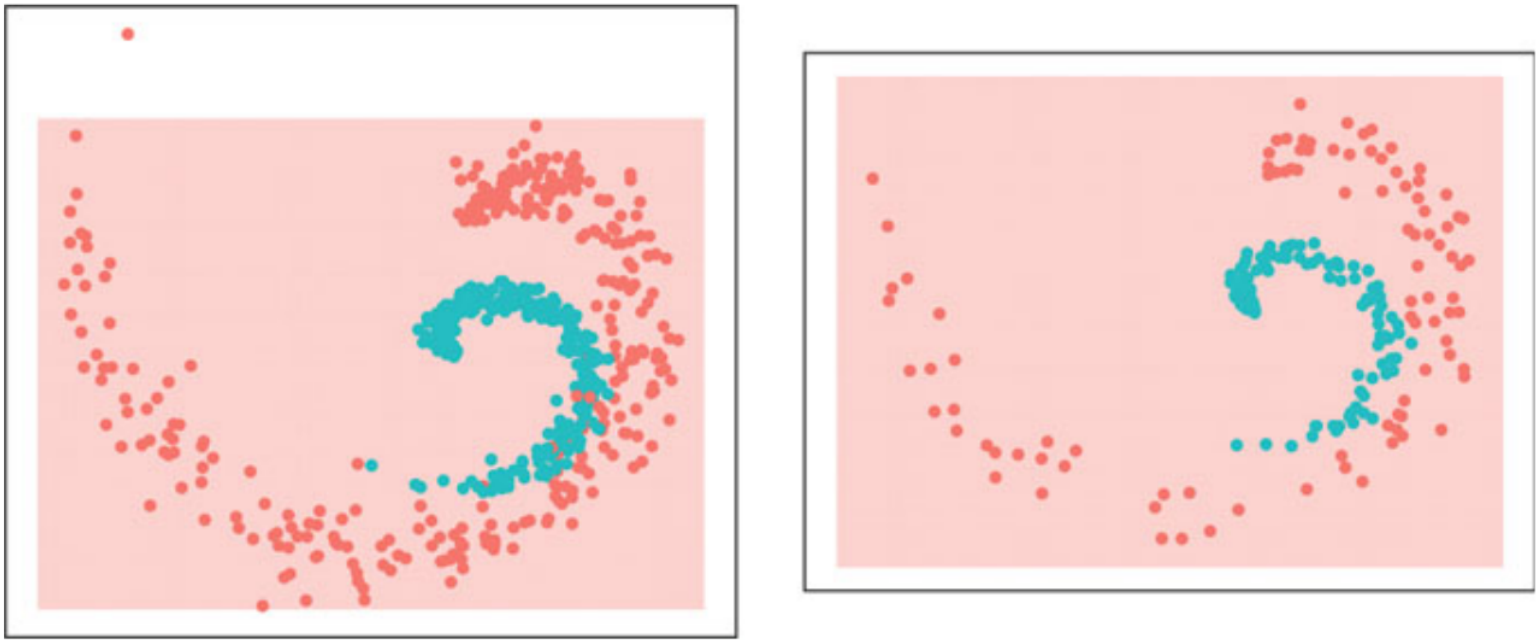


Figure: Decision boundary with zero initialization on the training data set (left) and the testing data set (right). The zero initialization does not break symmetry and the complexity of the network reduces to that of a single neuron (Ghatak, 2019).

WEIGHT INITIALIZATION: EXAMPLE / 3

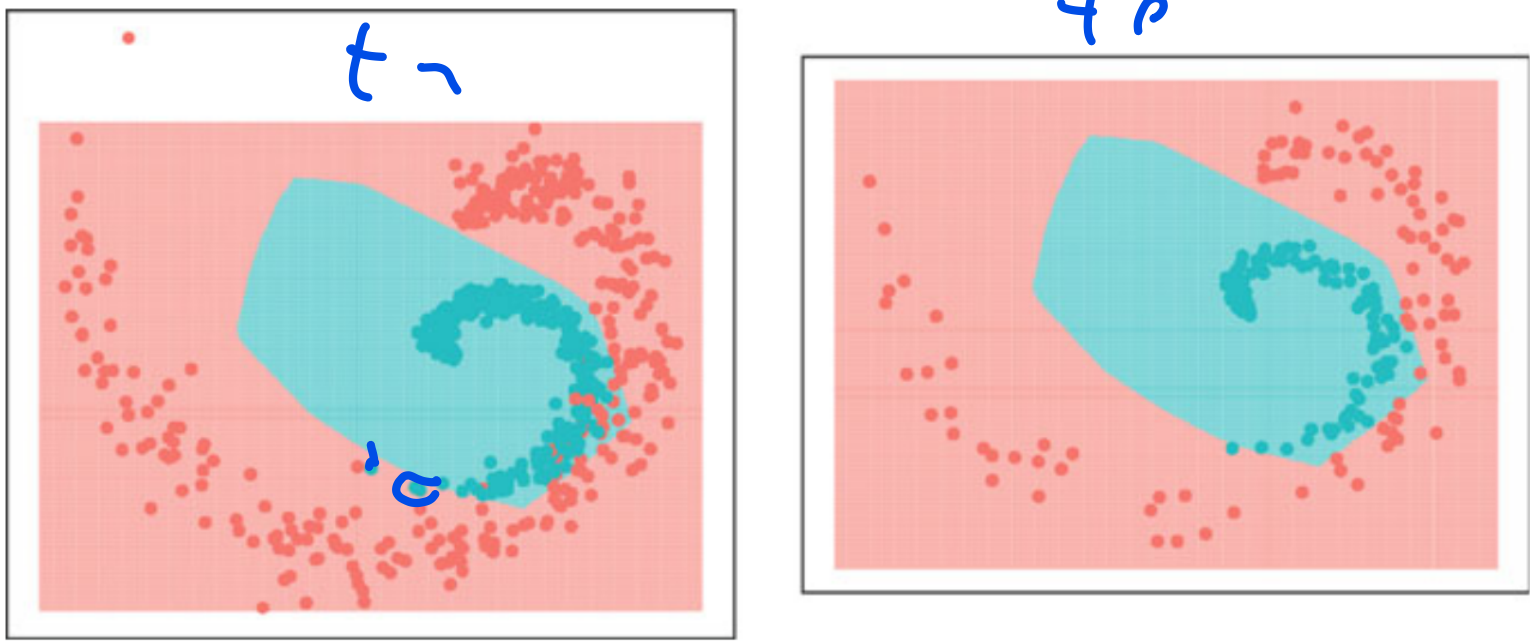


Figure: Decision boundary with random initialization ($N\left(0, 1 \cdot 10^{-4}\right)$) on the training data set (left) and the testing data set (right) (Ghatak, 2019).

WEIGHT INITIALIZATION: EXAMPLE / 4

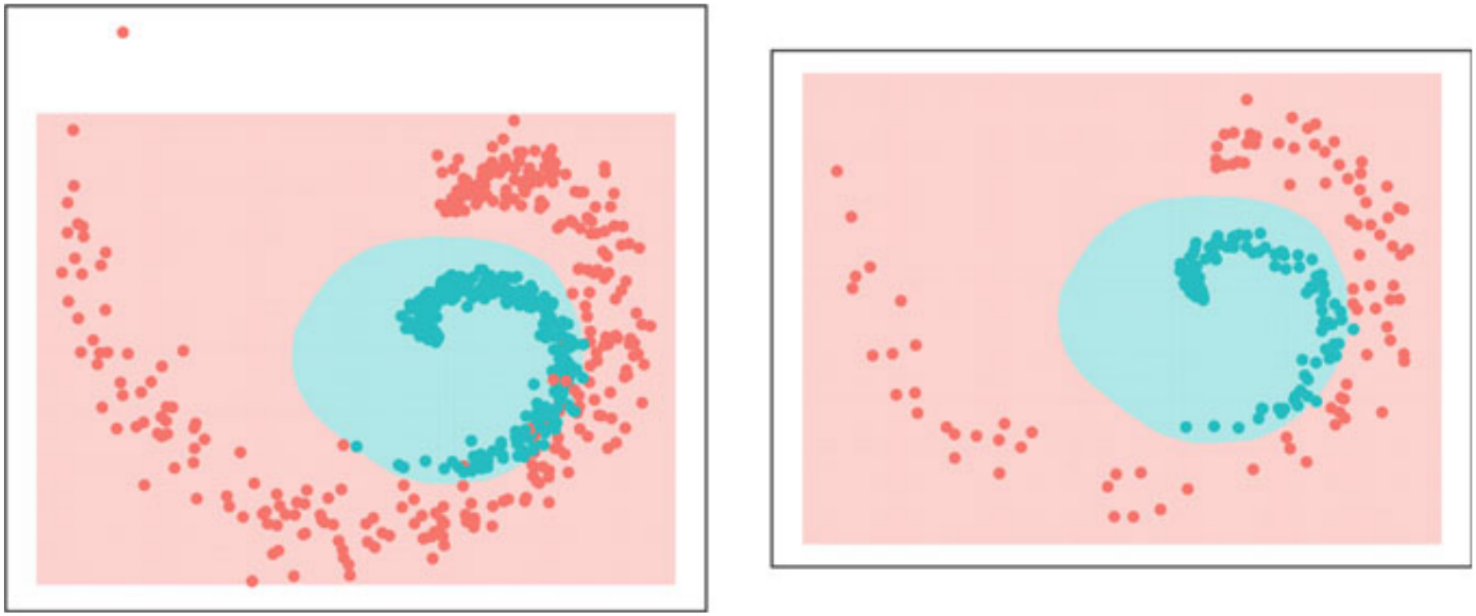


Figure: Decision boundary with He initialization on the training data set (left) and the testing data set (right) (Ghatak, 2019).

BIAS INITIALIZATION

- Typically, we set the biases for each unit to heuristically chosen constants.
- Setting the biases to zero is compatible with most weight initialization schemes as the schemes expect a small bias.
- However, deviations from 0 can be made individually, for example, in order to obtain the right marginal statistics of the output unit or to avoid causing too much saturation at the initialization.
- For details see Goodfellow et. al (2016).

REFERENCES


Ghatak, A. (2019). *Deep learning with R*. Springer Singapore.


Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Where He's $\mathrm{Var}(w) = 2/m$ comes from

One ReLU layer with m inputs: $z = \sum_{i=1}^{m} w_i\, a_i$, where $a_i = \max(0, z_i^{\text{prev}})$. Weights are i.i.d. with mean 0 and independent of the inputs; z^{prev} is symmetric around 0.

$$\mathrm{Var}(z) = \sum_{i=1}^{m} \mathrm{Var}(w_i a_i) = m\,\mathbb{E}[w^2]\,\mathbb{E}[a^2] = m\,\mathrm{Var}(w)\,\mathbb{E}[a^2] \tag{1}$$

$$\mathbb{E}[a^2] = \mathbb{E}\big[(z^{\text{prev}})^2\,\mathbb{1}\{z^{\text{prev}} > 0\}\big] = \tfrac{1}{2}\,\mathrm{Var}(z^{\text{prev}}) \tag{2}$$

$$\mathrm{Var}(z) = \tfrac{m}{2}\,\mathrm{Var}(w)\,\mathrm{Var}(z^{\text{prev}}) \tag{3}$$

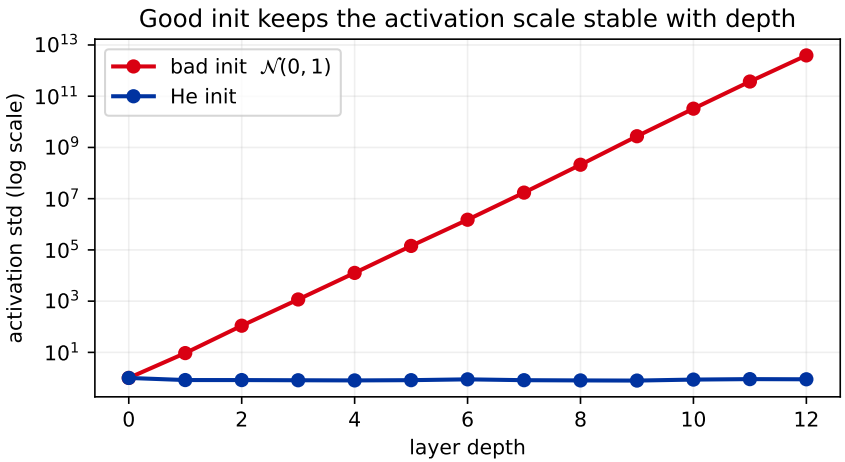
(1): mean-zero weights make the cross terms vanish. (2): ReLU zeroes the negative half of a symmetric z^{prev}.

Keep the scale fixed from layer to layer:
$\frac{m}{2}\,\mathrm{Var}(w) = 1 \;\Rightarrow\; \boxed{\mathrm{Var}(w) = 2/m}$ *(He et al., 2015)*.

Glorot assumes no ReLU halving: $1/m$ keeps the forward pass stable and $1/n$ the backward pass, so the compromise is $2/(m+n)$. LMU's uniform $\big[-\sqrt{6/(m+n)}, \sqrt{6/(m+n)}\big]$ has exactly that variance, since a uniform on $[-c, c]$ has variance $c^2/3$.

The formula predicts the figure

An experiment: push unit-variance input through 12 ReLU layers of width $m = 256$ and measure the spread of the activations at every layer.



With $\mathcal{N}(0,1)$ weights, (3) multiplies the variance by $m/2 = 128$ per layer, so the scale grows about $\sqrt{128} \approx 11\times$ per layer.

layer 12	**predicted**	**measured**
$\mathcal{N}(0,1)$	$3.6 \cdot 10^{12}$	$3.9 \cdot 10^{12}$
He	0.83	0.89

Std of the ReLU outputs; a ReLU output's std is $0.58\times$ that of its input.

Three lines of algebra get the blow-up right across twelve orders of magnitude (six seeds: 2.7 to $6.5 \cdot 10^{12}$). Other activations need their own gain: tanh squashes too, so PyTorch's `calculate_gain` gives 5/3. Karpathy's caveat: BatchNorm, residual connections and Adam make exact initialization much less critical than it once was (2022).

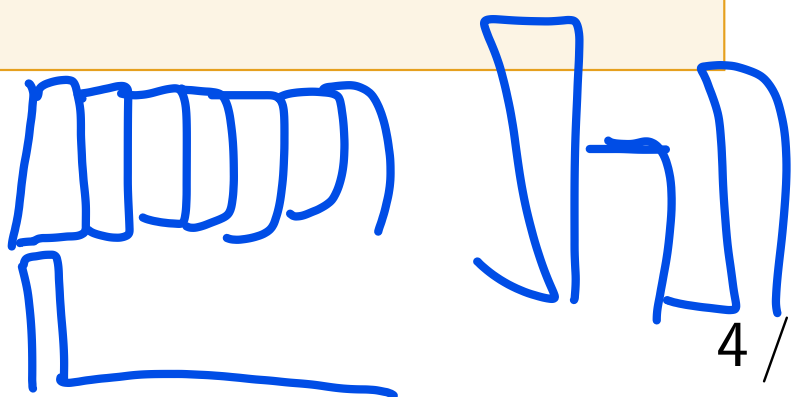
Predict: the loss before training starts

Your homework MLP classifies Fashion-MNIST into 10 classes with the cross-entropy loss.

You have built the network but not trained it.

What is its loss on the first batch?

Hint: a well-initialized network knows nothing yet. What probability should it give the correct class?



Answer: about $\ln 10 \approx 2.30$

Knowing nothing, the net should spread probability evenly: $p_{\text{correct}} \approx \frac{1}{10}$, so the loss is $-\ln \frac{1}{10} = \ln 10 \approx 2.30$. With C classes it is $\ln C$.

untrained MLP $(784 \to 256 \to 128 \to 10)$	**loss**	**logit std**	**mean top prob.**
PyTorch default init	2.31	0.07	0.11
all weights $\mathcal{N}(0, 1)$	1433	1008	0.998

Measured on 2000 Fashion-MNIST test images, before a single update. PyTorch's default `nn.Linear` weights are $U(-1/\sqrt{m}, 1/\sqrt{m})$: small, so the logits start near 0.

Karpathy's makemore net (27 characters) started at a loss of 27 instead of $\ln 27 \approx 3.30$. Its loss curve was a *hockey stick*: the first steps did nothing but shrink the logits (Karpathy, 2022).

A free sanity check: print the loss before training. Near $\ln C$ is good. Far above it means the net is *confidently wrong* from the start (bad init, unscaled inputs), and early training is spent just undoing that (Karpathy, 2019).

The activations as equations (1/2): sigmoid and tanh

Sigmoid

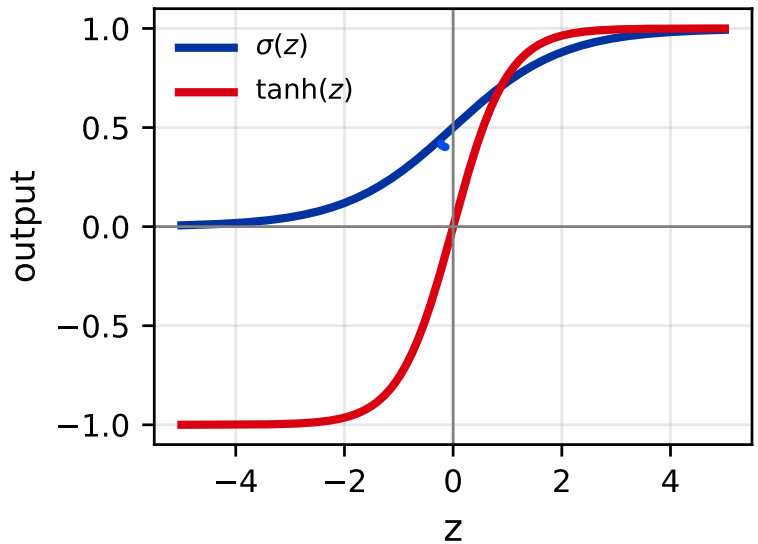
$$\sigma(z) = \frac{1}{1+e^{-z}} \in (0,1)$$

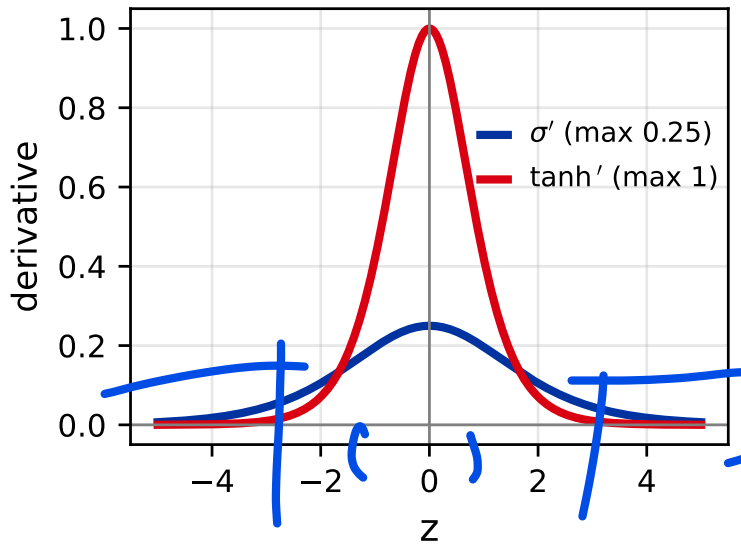
$$\sigma'(z) = \sigma(z)\left(1-\sigma(z)\right) \leq \tfrac{1}{4}$$

tanh

$$\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \in (-1,1)$$

$$\tanh'(z) = 1-\tanh^{2}(z) \leq 1$$

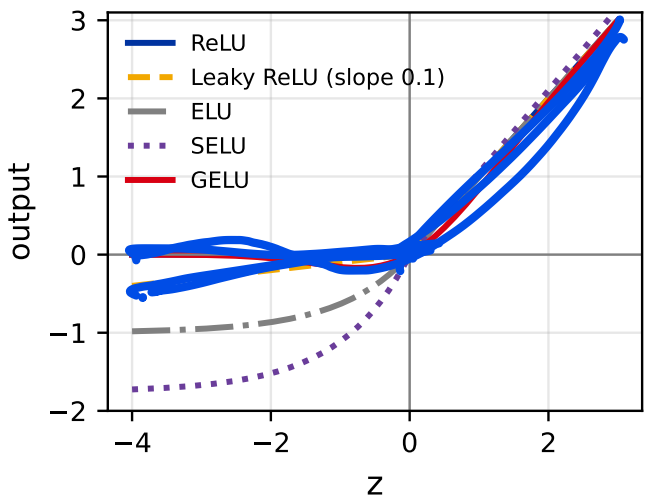


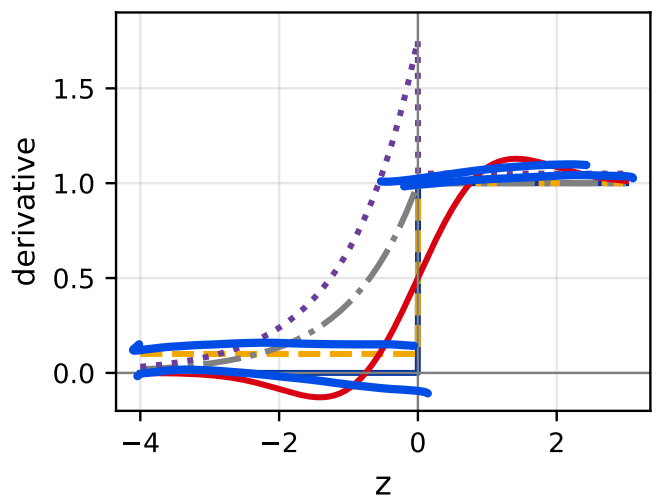


tanh is a stretched, shifted sigmoid: $\tanh(z) = 2\,\sigma(2z) - 1$. Its outputs are centred on 0 and its slope at 0 is 1, not $\frac{1}{4}$ – why tanh was preferred over sigmoid in hidden layers (LeCun et al., 1998). Both go flat for large $|z|$: the saturation behind vanishing gradients.

The activations as equations (2/2): ReLU family, GELU

	$f(z)$	$f'(z)$	**range**
ReLU	$\max(0, z)$	1 if $z > 0$, else 0	$[0, \infty)$
Leaky ReLU	z if $z \geq 0$, else az $(a = 0.01)$	1 or a	$(-\infty, \infty)$
PReLU	Leaky ReLU with a *learned*	1 or a	$(-\infty, \infty)$
ELU	z if $z \geq 0$, else $a\,(e^z - 1)$ $(a = 1)$	1 or $a\,e^z$	$(-a, \infty)$
SELU	$\lambda \cdot \mathrm{ELU}_a(z)$, $a \approx 1.673$, $\lambda \approx 1.051$	λ or $\lambda a\,e^z$	$(-1.76, \infty)$
GELU	$z\,\Phi(z)$, Φ = standard normal CDF	$\Phi(z) + z\,\varphi(z)$	$[-0.17, \infty)$

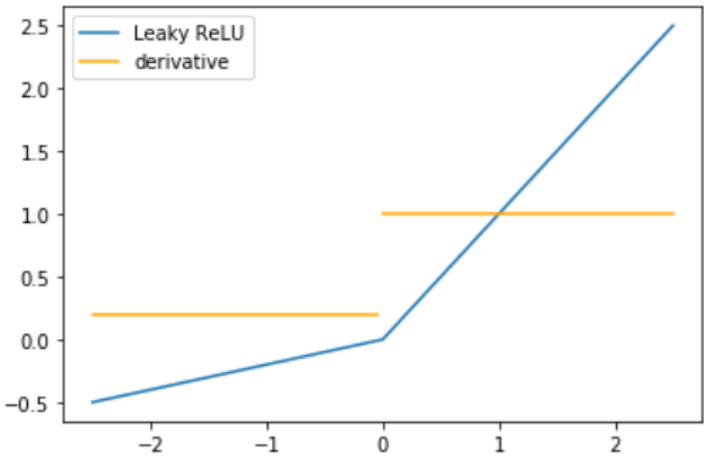




PyTorch's default constants; the plot uses $a = 0.1$ for Leaky ReLU so it is visible. Leaky ReLU (Maas et al., 2013); Parametric ReLU, PReLU (He et al., 2015); Exponential Linear Unit, ELU (Clevert et al., 2016); Scaled ELU, SELU (Klambauer et al., 2017); Gaussian Error Linear Unit, GELU (Hendrycks & Gimpel, 2016).

Deep Learning

Modern Activation Functions



Learning goals

- Challenges in Optimization related to Activation Functions
- Activations for Hidden Units
- Actications for Output Units

Hidden activations

HIDDEN ACTIVATIONS

- Recall, hidden-layer activation functions make it possible for deep neural nets to learn complex non-linear functions.
- The design of hidden units is an extremely active area of research. It is usually not possible to predict in advance which activation will work best. Therefore, the design process often consists of trial and error.
- In the following, we will limit ourselves to the most popular activations - Sigmoidal activation and ReLU.
- It is possible for many other functions to perform as well as these standard ones. An overview of further activations can be found ▸ here.

SIGMOIDAL ACTIVATIONS

- Sigmoidal functions such as *tanh* and the *logistic sigmoid* bound the outputs to a certain range by "squashing" their inputs.

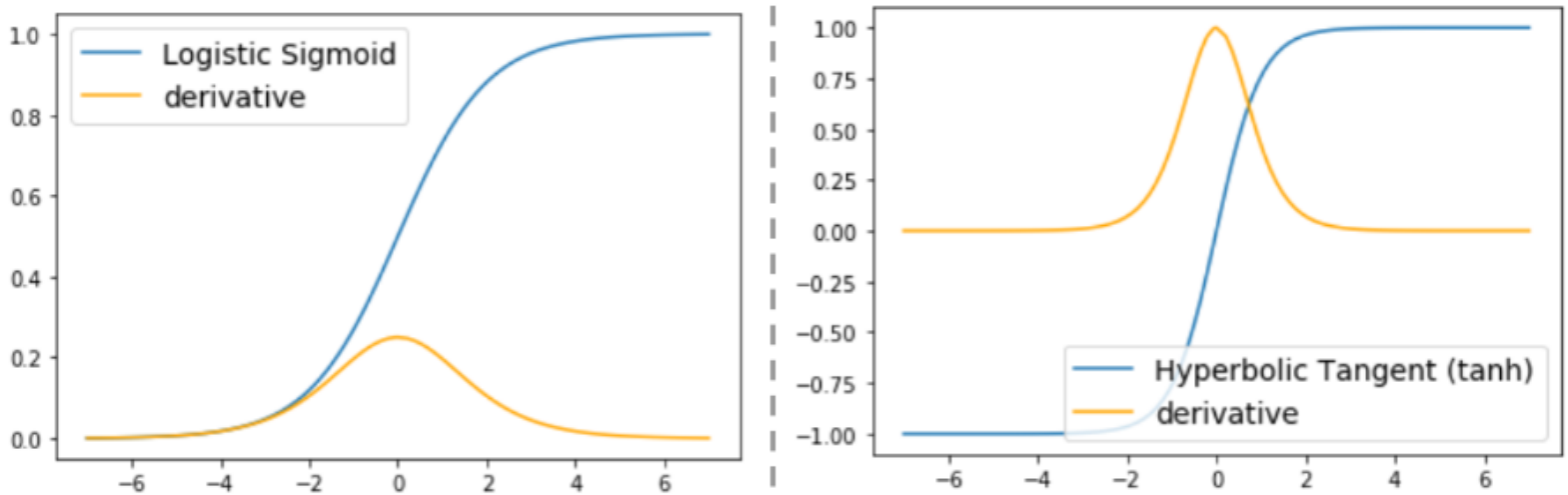


Figure: Sigmoidal activations (Savino & Tondolo, 2021)

- In each case, the function is only sensitive to its inputs in a small neighborhood around 0.
- Furthermore, the derivative is never greater than 1 and is close to zero across much of the domain.

SIGMOIDAL ACTIVATION FUNCTIONS

1. **Saturating Neurons:**
   - We know: $\sigma'(z_{in}) \to 0$ for $|z_{in}| \to \infty$.

   $\to$ Neurons with sigmoidal activations "saturate" easily, that is, they stop being responsive when $|z_{in}| \gg 0$.

SIGMOIDAL ACTIVATION FUNCTIONS / 2

2. **Vanishing Gradients:** Consider the vector of error signals $\delta^{(i)}$ in layer i

$$\delta^{(i)} = \mathbf{W}^{(i+1)}\delta^{(i+1)} \odot \sigma'\left(\boldsymbol{z}_{in}^{(i)}\right), i \in \{1, ..., O\}.$$

Each k-th component of the vector expresses how much the loss L changes when the input to the k-th neuron $z_{k,in}^{(i)}$ changes.

- We know: $\sigma'(z) < 1$ for all $z \in \mathbb{R}$.

$\rightarrow$ In each step of the recursive formula above, the value will be multiplied by a value smaller than one

$$\begin{aligned}
\delta^{(1)} &= \mathbf{W}^{(2)}\delta^{(2)} \odot \sigma'\left(z_{in}^{(i)}\right) \\
&= \mathbf{W}^{(2)}\left(\mathbf{W}^{(3)}\delta^{(3)} \odot \sigma'\left(z_{in}^{(i)}\right)\right) \odot \sigma'\left(z_{in}^{(i)}\right) \\
&= \ldots
\end{aligned}$$

- When this occurs, earlier layers train *very* slowly (or not at all).

RECTIFIED LINEAR UNITS (RELU)

- The ReLU activation solves the vanishing gradient problem.

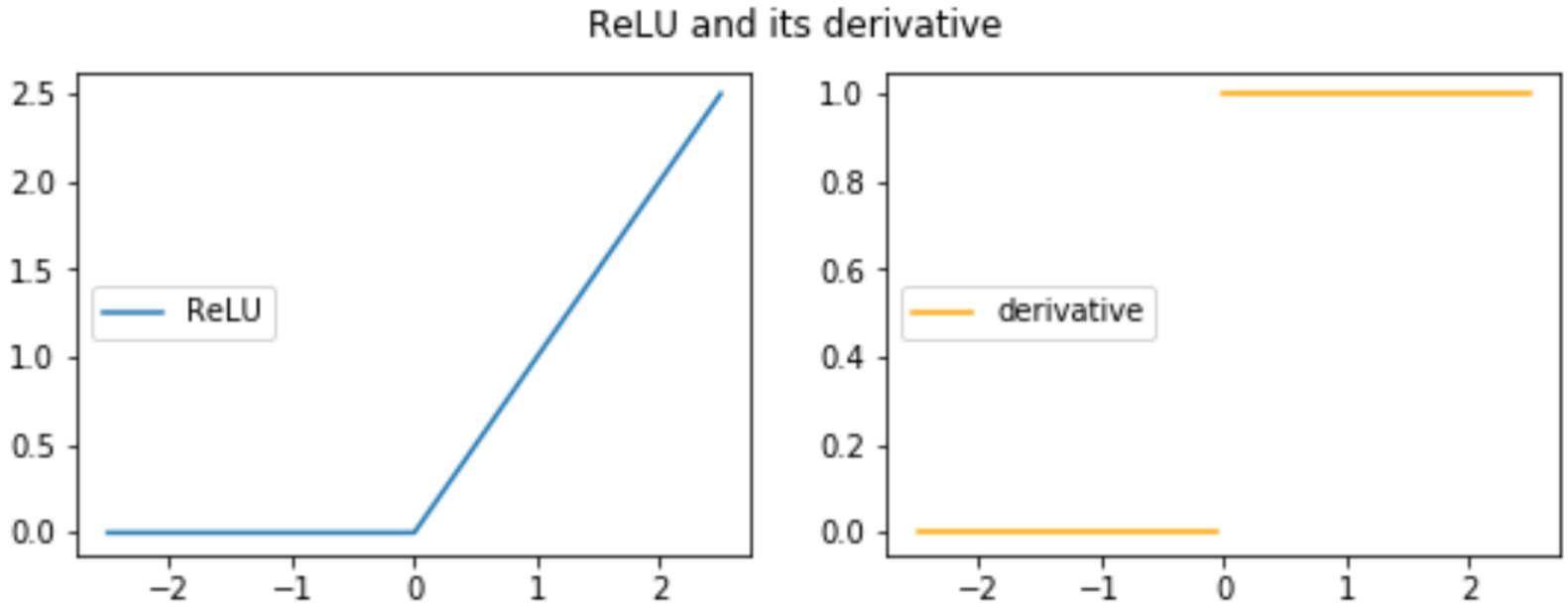


Figure: ReLU activation (Savino & Tondolo, 2021)

- In regions where the activation is positive, the derivative is 1.
- Derivatives do not vanish along paths containing "active" neurons.
- Note that the ReLU is not differentiable at 0 (Software implementations usually simply return 0).

RECTIFIED LINEAR UNITS (RELU)

- ReLU units can significantly speed up training compared to units with saturating activations.

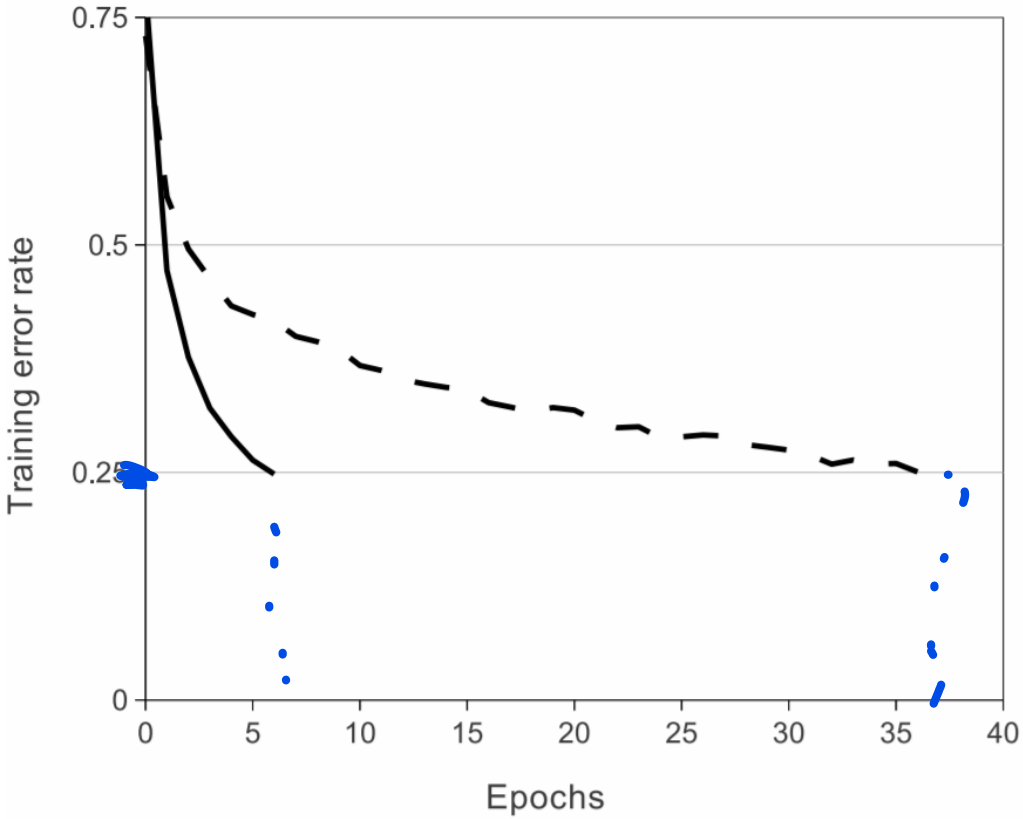


Figure: A four-layer convolutional neural network with ReLUs (solid line) reaches a 25% training error rate on the CIFAR-10 dataset six times faster than an equivalent network with tanh neurons (dashed line) (Krizhevsky et al., 2012).

RECTIFIED LINEAR UNITS (RELU)

- A downside of ReLU units is that when the input to the activation is negative, the derivative is zero (the "dying ReLU problem").

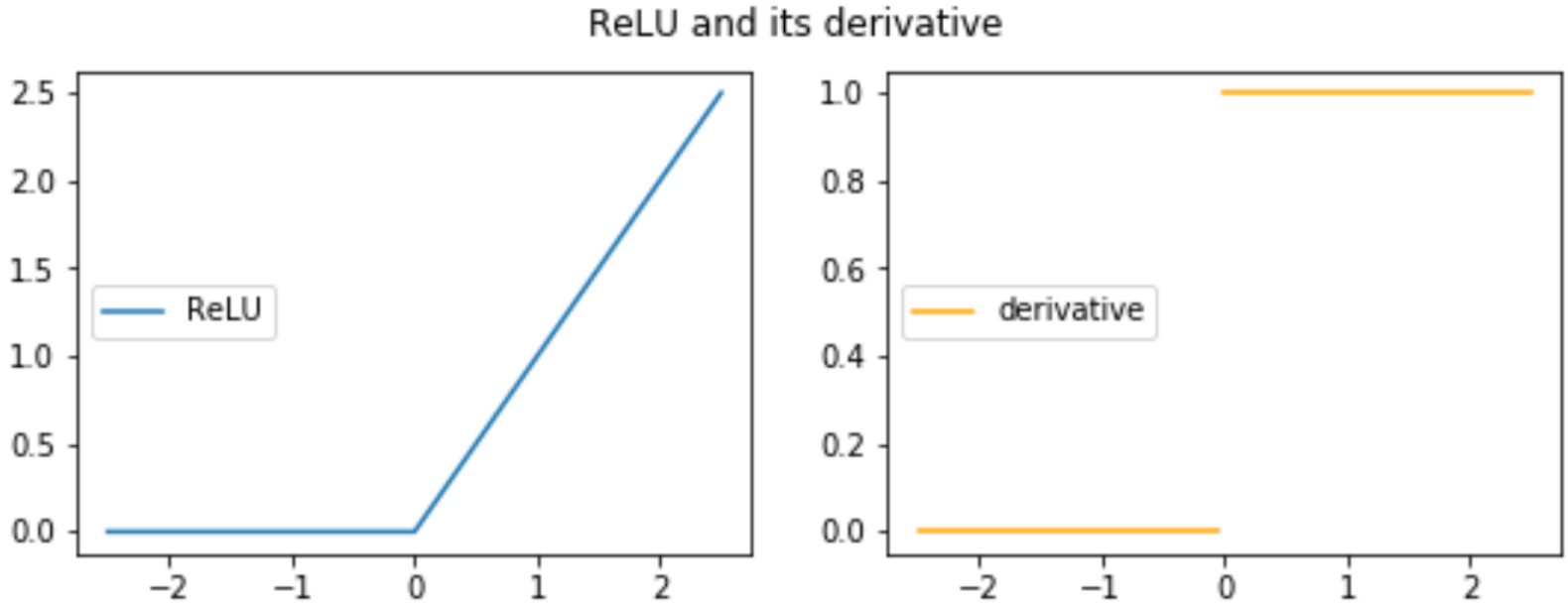


Figure: ReLU activation (Savino & Tondolo, 2021)

- When a ReLU unit "dies" (i.e. its activation is 0 for all datapoints), it kills the gradient flowing during backpropogation.
- Such units are never updated during training.

GENERALIZATIONS OF RELU

- There exist several generalizations of the ReLU activation that have non-zero derivatives throughout their domains.
- *Leaky ReLU*:

$$LReLU(v) = \begin{cases} v & v \geq 0 \\ \alpha v & v < 0 \end{cases}$$

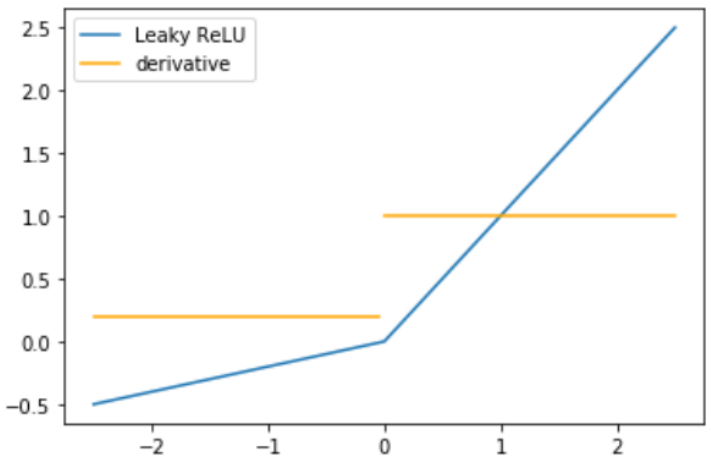


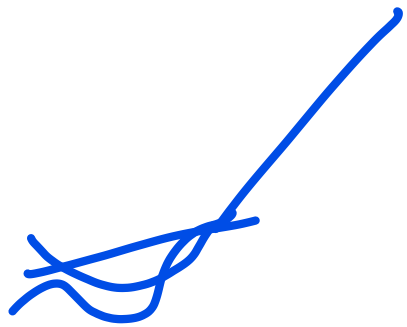
Figure: Leaky ReLU (Savino & Tondolo, 2021)

- Unlike the ReLU, when the input to the Leaky ReLU activation is negative, the derivative is α which is a small positive value.

GENERALIZATIONS OF RELU

- A variant of the Leaky ReLU is the *Parametric ReLU (PReLU)* which learns the α from the data through backpropagation.
- *Exponential Linear Unit (ELU)*:

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



- *Scaled Exponential Linear Unit (SELU)*:

$$SELU(v) = \lambda \begin{cases} v & v \geq 0 \\ \alpha(e^v - 1) & v < 0 \end{cases}$$

Note: In ELU and SELU, α and λ are hyperparameters that are set before training.

- These generalizations may perform as well as or better than the ReLU on some tasks.

GENERALIZATIONS OF RELU

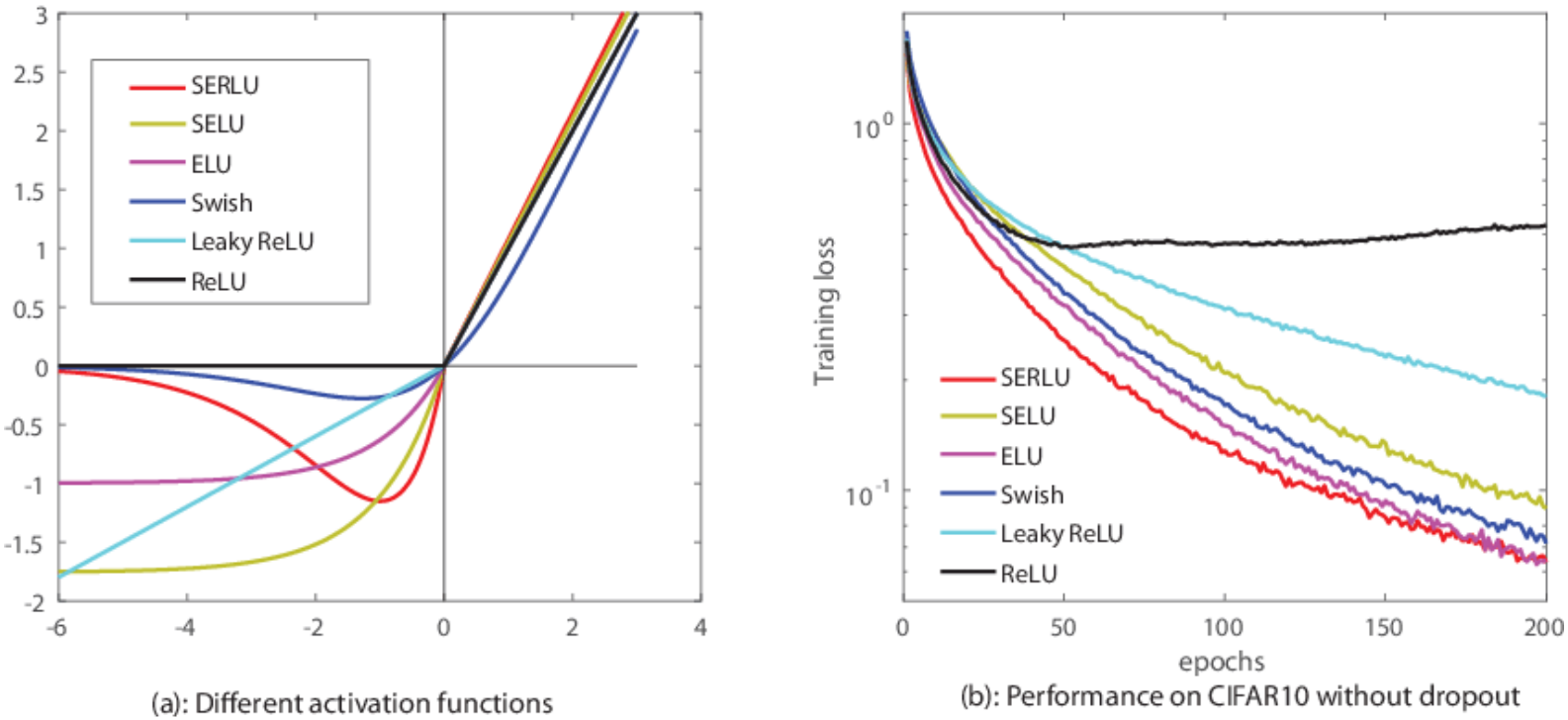


Figure: Visualization of different generalisations of ReLU (Zhang et al., 2018)

Output activations

OUTPUT ACTIVATIONS

- As we have seen previously, the role of the output activation is to get the final score on the same scale as the target.
- The output activations and the loss functions used to train neural networks can be viewed through the lens of maximum likelihood estimation (MLE).
- In general, the function $f(\mathbf{x} \mid \boldsymbol{\theta})$ represented by the neural network defines the conditional $p(y \mid \mathbf{x}, \boldsymbol{\theta})$ in a supervised learning task.
- Maximizing the likelihood is then equivalent to minimizing $-\log p(y \mid \mathbf{x}, \boldsymbol{\theta})$.
- An output unit with the identity function as the activation can be used to represent the mean of a Gaussian distribution.
- For such a unit, training with mean-squared error is equivalent to maximizing the log-likelihood (ignoring issues with non-convexity).

OUTPUT ACTIVATIONS

- Similarly, sigmoid and softmax units can output the parameter(s) of a Bernoulli distribution and Categorical distribution, respectively.
- It is straightforward to show that when the label is one-hot encoded, training with the cross-entropy loss is equivalent to maximizing log-likelihood. ▸ Click here
- Because these activations can saturate, an important advantage of maximizing log-likelihood is that the log undoes some of the exponentiation in the activation functions which is desirable when optimizing with gradient-based methods.
- For example, in the case of softmax, the loss is:

$$L(y, f(\mathbf{x})) = -f_{in,k} + \log \sum_{k'=1}^{g} \exp(f_{in,k'})$$

where k is the correct class. The first term, $-f_{in,k}$, does not saturate which means training can progress steadily even if the contribution of $f_{in,k}$ to the second term is negligible.

OUTPUT ACTIVATIONS

- A neural network can even be used to output the parameters of more complex distributions.
- A Mixture Density Network, for example, outputs the parameters of a Gaussian Mixture Model:

$$p(y|\mathbf{x}) = \sum_{c=1}^{m} \phi^{(c)}(\mathbf{x})\, \mathcal{N}\left(y; \boldsymbol{\mu}^{(c)}(\mathbf{x}), \boldsymbol{\Sigma}^{(c)}(\mathbf{x})\right)$$

  where m the number of components in the mixture.
- In such a network, the output units are divided into groups.
- One group of output neurons with softmax activation represents the weights ($\phi^{(c)}$) of the mixture.
- Another group with the identity activation represents the means ($\boldsymbol{\mu}^{(c)}$) and yet another group with a non-negative activation function (such as ReLU or the exponential function) can represent the variances of the (typically) diagonal covariance matrices $\boldsymbol{\Sigma}^{(c)}$.

OUTPUT ACTIVATIONS

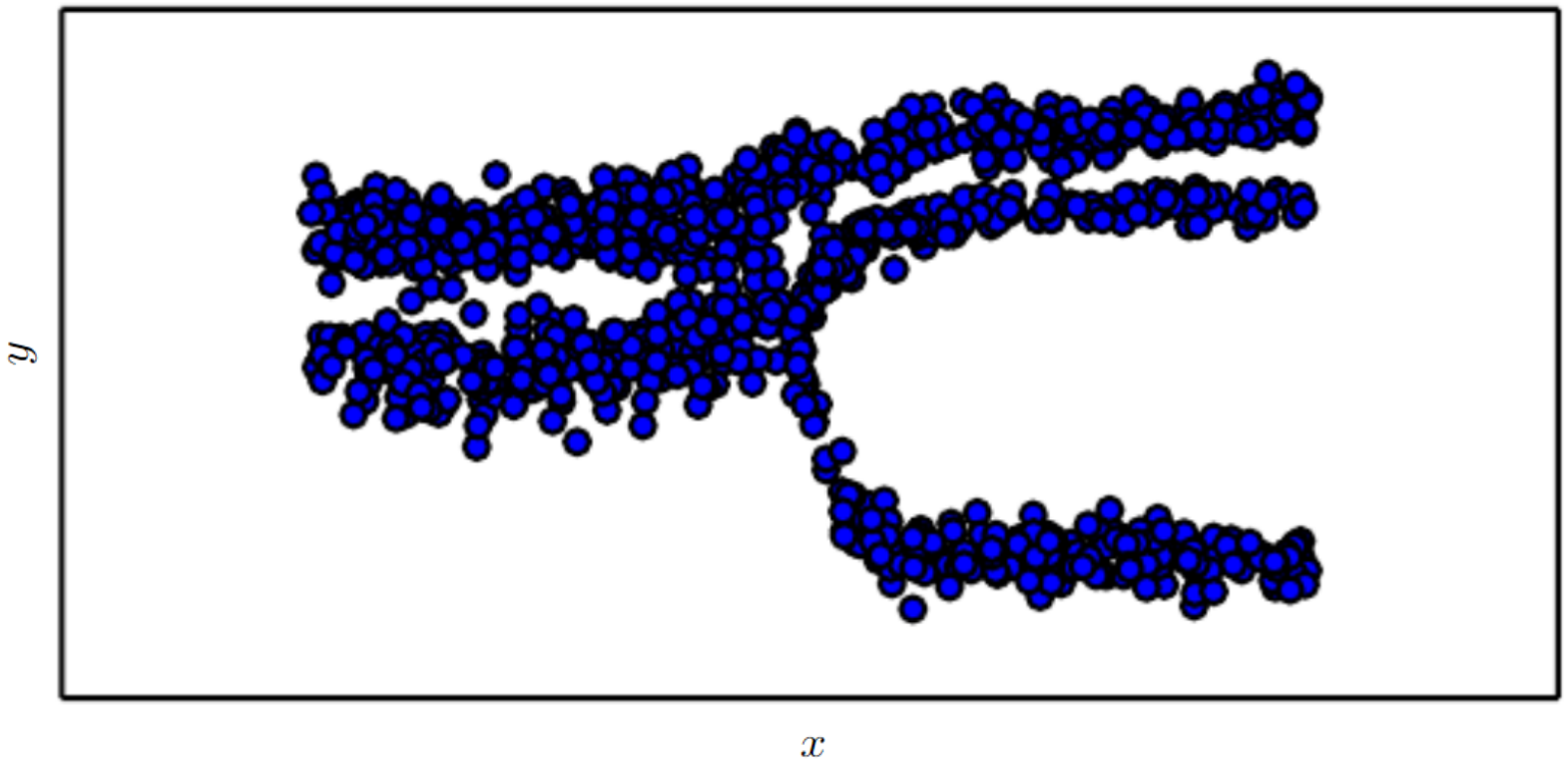


Figure: Samples drawn from a Mixture Density Network. The input **x** is sampled from a uniform distribution and y is sampled from $p(y \mid \mathbf{x}, \boldsymbol{\theta})$ (Goodfellow et al., 2016).

REFERENCES

- Savino, P., & Tondolo, F. (2021). Automated classification of civil structure defects based on convolutional neural network. *Frontiers of Structural and Civil Engineering, 15*(2), 305–317. https://doi.org/10.1007/s11709-021-0725-9
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems* (Vol. 25). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
- Zhang, G., & Li, H. (2018). *Effectiveness of Scaled Exponentially-Regularized Linear Units (SERLUs)*.

Which activation? The short answer

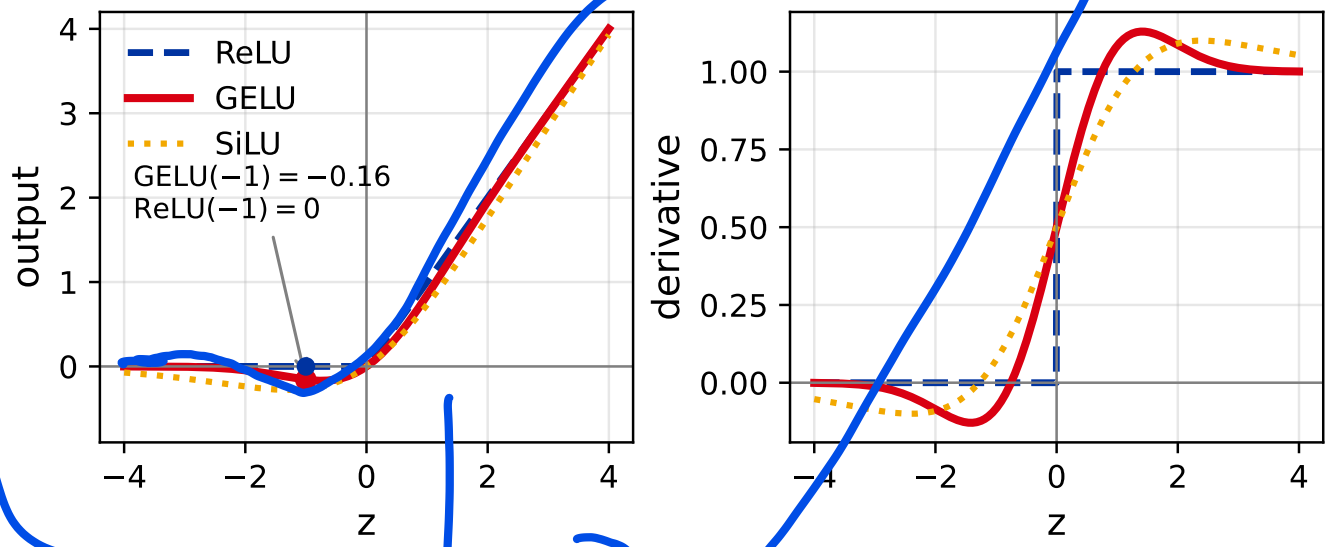
LMU showed the full zoo – sigmoid, tanh, ReLU, Leaky ReLU, Parametric ReLU (PReLU), Exponential Linear Unit (ELU) and Scaled ELU (SELU). In practice:

- **Hidden layers: ReLU by default.** Its derivative is 1 for $z > 0$, so gradients don't vanish, and it is cheap. Reach for **Leaky ReLU** or **ELU** only if you hit *dying ReLUs*. Transformers instead use the smooth **Gaussian Error Linear Unit (GELU)** – next frame.
- **Output layer: match the task – identity** for regression, **sigmoid** for binary, **softmax** for multiclass.
- **Sigmoid/tanh in hidden layers: avoid** – they saturate and vanish the gradient (the backprop recursion from the training deck, made concrete).

Output activation + loss are two sides of one coin (maximum likelihood): sigmoid+cross-entropy, softmax+cross-entropy, identity+MSE – which is why the output gradient is the clean residual $f - y$.

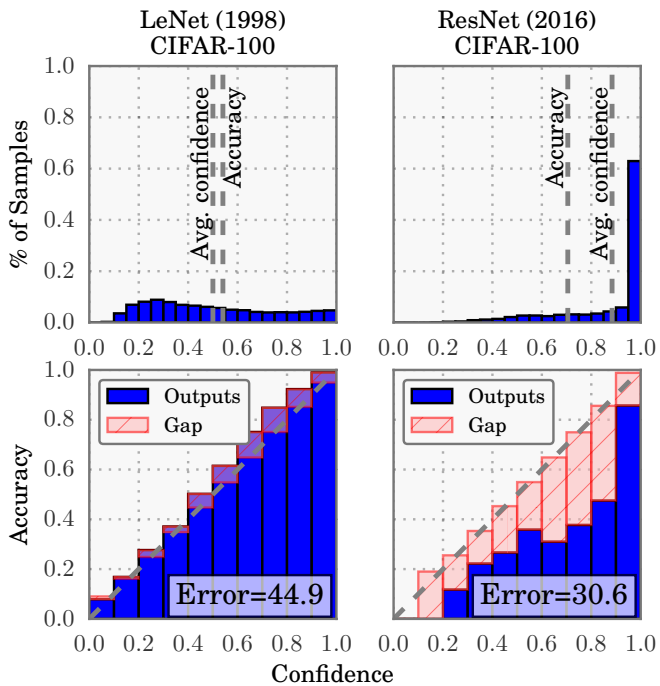
GELU: the activation transformers actually use

GELU (Hendrycks & Gimpel, 2016): $\mathrm{GELU}(z) = z\,\Phi(z)$, with Φ the standard normal CDF. It *weights* an input by its value instead of *gating* it by its sign, the way ReLU does.



Why it helps. It is smooth, and its derivative is *non-zero* for negative inputs: at $z = -1$ GELU returns -0.159 with slope -0.083, where ReLU returns 0 with slope 0 – so a unit that drifts negative is not dead. Used in **BERT** and **GPT-2/3**; today's open models prefer the gated **SwiGLU** (Shazeer, 2020) – PaLM, Llama. Both return in the LLM chapter.

Modern networks are overconfident



Guo et al. (ICML 2017), Fig. 1, CIFAR-100

Reliability diagrams, as in the calibration lecture. A 5-layer **LeNet (1998)** is about as confident as it is accurate. A 110-layer **ResNet (2016)** is more *accurate* (error 44.9% $\rightarrow$ 30.6%), but its confidence piles up near 1 and its accuracy does not.

The paper names depth, width, weight decay and BatchNorm as key factors – the tools this chapter just taught.

Confidence is not competence. Softmax must spread probability over the classes it knows, so a net shown pure noise still has to pick one, and nothing stops it being confident. **The cheap fix:** temperature scaling, a one-parameter Platt scaling. Divide the logits by one number T fitted on validation data; the argmax, and so the accuracy, is unchanged.

Two silent bugs the homework will hit

1. Unscaled inputs. Trees never cared about feature scale; neural nets do. A feature measured in thousands dominates the gradient and makes the loss surface badly conditioned (the ill-conditioning from part 1). *Fix:* standardize, or scale pixels to $[0, 1]$, fitting the scaler on the training split only.

2. Forgetting `model.eval()`. Dropout and BatchNorm behave differently at test time:

	dropout	**BatchNorm**
`model.train()`	on	statistics of the current batch
`model.eval()`	off	running averages from training

Forget it and test predictions come out noisy and depend on what else is in the batch. Note that `eval()` does not switch off gradient tracking – that is `torch.no_grad()`.

Neither bug raises an error. The code runs and the numbers are just worse – worth knowing before the homework, not after.

Before a long training run: four cheap checks

Condensed from Karpathy's *A Recipe for Training Neural Networks* (2019) and his makemore lectures (2022):

1. **Loss at step 0** $\approx \ln C$ (the prediction frame earlier). If not, fix the initialization or the input scaling before anything else.
2. **Overfit one tiny batch.** Train on a handful of examples until the loss is near 0. A network that cannot memorize a handful of examples has a bug, not a regularization problem.
3. **Plot train and validation loss every epoch** (regularization deck), and only then start turning knobs, one at a time.
4. **Watch the update size.** For each weight matrix, $\mathrm{std}(\alpha \nabla W)/\mathrm{std}(W) \approx 10^{-3}$ is Karpathy's rule of thumb. Far below it, the learning rate is too low; far above, too high.

Each check takes a minute and catches the bugs that never raise an error. Run all four before the homework's long runs.

The neural-networks chapter, end to end

One machine: a neuron, stacked, trained, regularized, and optimized.

The whole chapter in one arc

- **A neuron** = affine + activation = a regression you know. **Hidden layers** learn a feature space so the boundary can bend (representation learning).
- **Depth & width** make it expressive (universal approximation, folding); **softmax** handles many classes.
- **Backprop** = the chain rule backward once, reusing shared error signals – it gives every gradient in a single pass.
- **Regularization** (weight decay, early stopping, dropout, augmentation) fights overfitting; **optimization** (SGD+momentum / Adam, schedules, init, BatchNorm) makes training fast and stable.

Every giant model from the intro deck – LLMs, diffusion, AlphaFold – is this same machinery, scaled up and specialized. You now know how it is built and trained.

Where next: architectures shaped to the data

The MLP treats every input feature the same. Real signals have **structure** a plain MLP wastes:

- **Images** have *locality* and *translation* structure → **convolutional networks (CNNs)**.
- **Sequences** (text, audio, time series) have *order* and *memory* → **recurrent nets / transformers**.

Same neurons, same backprop, same optimizers – but the *architecture* is shaped to the data. **Next chapter: convolutional neural networks.**