# Deep Learning

## Lecture 4 – Deep Neural Networks II

Sebastian Bordt
Theory of Machine Learning Group
University of Tübingen









# Agenda

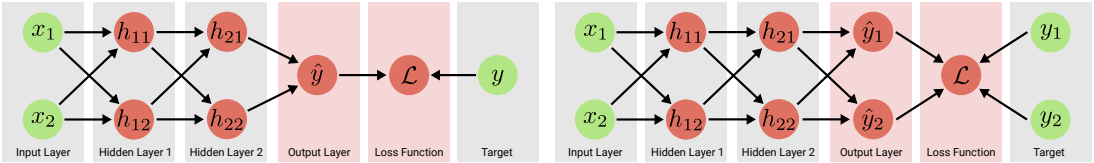
**4.1** Output and Loss Functions

**4.2** Preprocessing and Initialization

**4.3** Data Augmentation

# 4.1
# Output and Loss Functions

# Output and Loss Functions

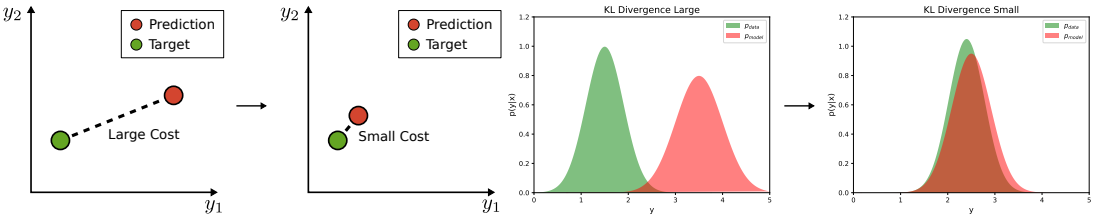



- The **output layer** is the last layer in a neural network which computes the output
- The **loss function** compares the result of the output layer to the target value(s)
- Choice of output layer and loss function depends on task (discrete, continuous, ..)

# Loss Function

What is the goal of optimizing the loss function?

- Tries to make the **model output** (=prediction) similar to the **target** (=data)
- Think of the loss function as a **measure of cost** being paid for a prediction



# Loss Function

How to design a good loss function?

- A loss function can be any differentiable function that we wish to optimize
- Deriving the cost function from the **maximum likelihood principle** removes the burden of manually designing the cost function for each model
- Consider the output of the neural network as **parameters of a distribution** over $y_i$

$$\begin{aligned}\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}}\ p_{model}(\mathbf{y}|\mathbf{X}, \mathbf{w}) \\ &\overset{\text{iid}}{=} \underset{\mathbf{w}}{\operatorname{argmax}} \prod_{i=1}^{N} p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\ &= \underset{\mathbf{w}}{\operatorname{argmax}} \underbrace{\sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w})}_{\text{Log-Likelihood}}\end{aligned}$$

# Recap: Loss Functions from Lecture 2

**Maximum likelihood:** the loss is the **negative log-likelihood** $-\log p_{model}(y|\mathbf{x}, \mathbf{w})$

**Regression,** $y \in \mathbb{R}$

Gaussian $\Rightarrow$ **squared loss**

$$\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$$

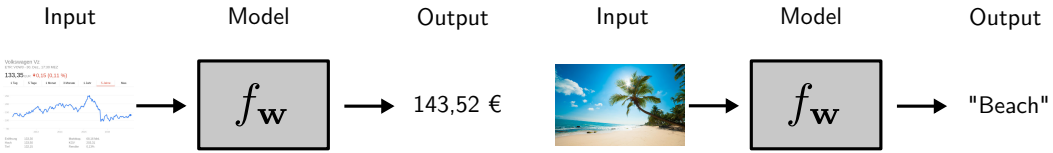
**Binary classification,** $y \in \{0, 1\}$

Bernoulli $\Rightarrow$ **binary cross-entropy**

$$\mathcal{L}(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

In lecture 2, $\hat{y} = f_{\mathbf{w}}(\mathbf{x})$ was linear; now it is an MLP, with the same losses.

# Recap: Regression and Classification




- **Mapping:**
  - Regression: $f_{\mathbf{w}} : \mathbb{R}^N \rightarrow \mathbb{R}$
  - Binary Classification: $f_{\mathbf{w}} : \mathbb{R}^{W \times H} \rightarrow \{\text{“Beach”}, \text{“No Beach”}\}$
  - Multi-Class Classification: $f_{\mathbf{w}} : \mathbb{R}^{W \times H} \rightarrow \{\text{“Beach”}, \text{“Mountain”}, \text{“City”}, \text{“Forest”}\}$

# Regression Problems

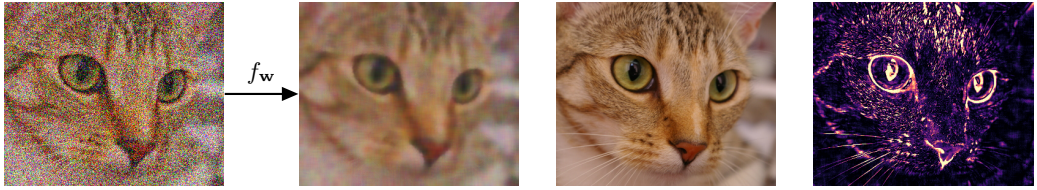
# Regression with Many Outputs

- Often the target is not one number but many: $\mathbf{y} \in \mathbb{R}^D$,
  e.g., an image with $D = H \times W \times 3$ values
- An **independent Gaussian** for each output $d$, with mean $\hat{y}_d$
- The negative log-likelihood is a **sum of squared errors:**

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{d=1}^{D} (\hat{y}_d - y_d)^2 = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

- For images: the **pixel-wise** $L_2$ **loss**

# Example: Image Denoising




input $\mathbf{x}$:
noisy image

prediction
$\hat{\mathbf{y}} = f_{\mathbf{w}}(\mathbf{x})$

target $\mathbf{y}$:
clean image

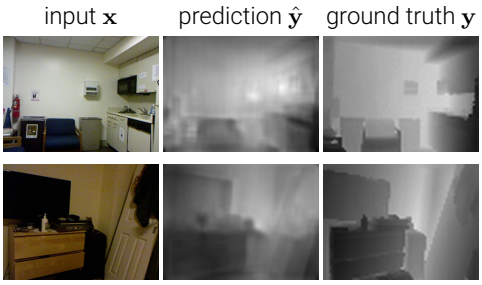
squared error
per pixel

- The output is an **image**: compare it with the ground-truth image, pixel by pixel

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{D}\sum_{p=1}^{D}(\hat{y}_p - y_p)^2 \qquad \text{(pixel-wise } L_2 \text{ loss, MSE)}$$

- Training data is easy to make: take clean images and add noise

# Example: Depth Estimation

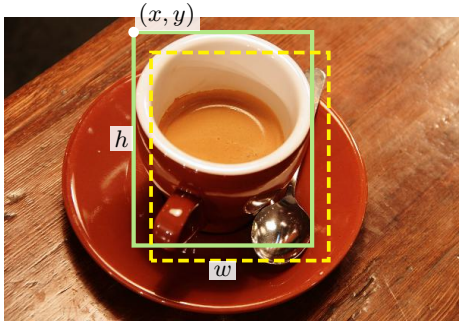



- Predict the **depth** of each pixel
- Output: a **depth map** (an image)
- Loss: per-pixel $L_2$ on log-depth

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n} \sum_{p=1}^{n} (\log \hat{y}_p - \log y_p)^2$$

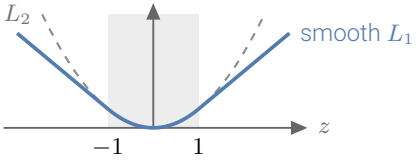
Eigen, Puhrsch and Fergus: Depth Map Prediction from a Single Image using a Multi-Scale Deep Network. NIPS, 2014.

# Example: Bounding Boxes




— ground truth $\mathbf{y}$ - - prediction $\hat{\mathbf{y}}$

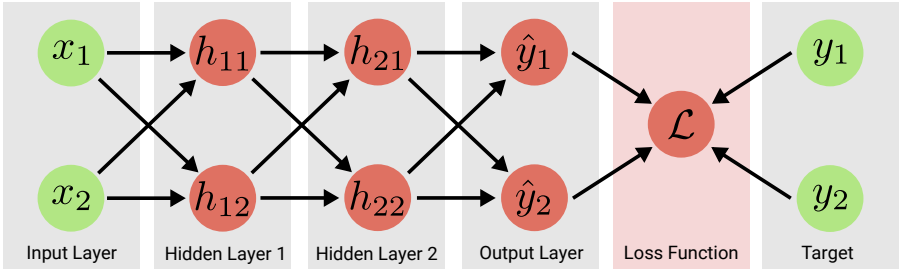
- **Object detection:** box around each object
- A box is 4 numbers $(x, y, w, h)$: $D = 4$
- Fast R-CNN: **smooth $L_1$ loss** on each error $z = \hat{y}_d - y_d$




$\frac{1}{2}z^2$ for $|z| < 1$, then linear: $|z| - \frac{1}{2}$

Girshick: Fast R-CNN. ICCV, 2015. Image: scikit-image sample data (CC0).

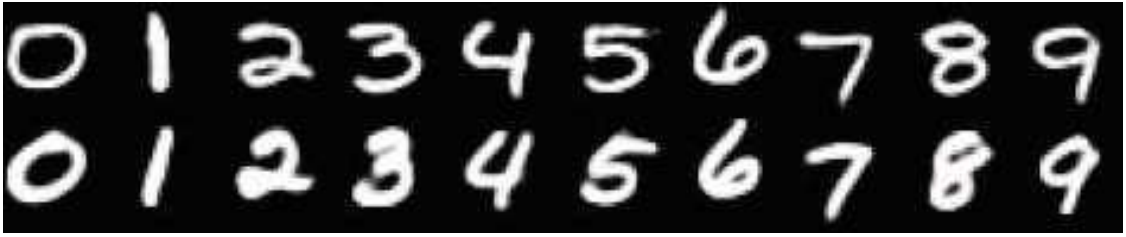
# Loss Function for Regression Problems




| **Task** | **Output $\hat{\mathbf{y}}$** | **Loss** |
|---|---|---|
| Regression (lecture 2) | one number | squared ($L_2$) |
| Image denoising | image | pixel-wise $L_2$ (MSE) |
| Depth estimation | depth map | per-pixel $L_2$ on log-depth |
| Bounding boxes | 4 numbers | smooth $L_1$ |

# Classification Problems

# Image Classification

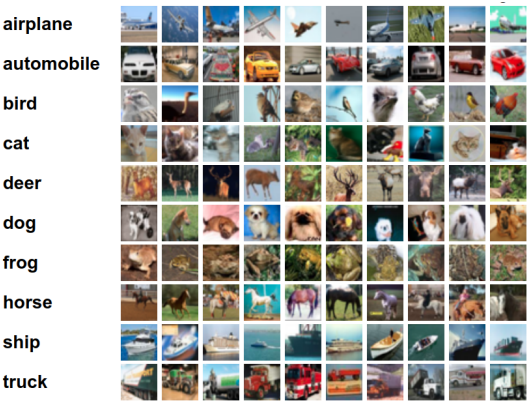


**MNIST Handwritten Digits:**

- Historically, one of the most popular datasets in ML
- Based on data from the National Institute of Standards and Technology
- Hand written by Census Bureau employees and high-school children
- Resolution: 28 x 28 pixels, 60k training samples with labels, 10k test samples

LeCun, Bottou, Bengio and Haffner: Gradient-based learning applied to document recognition. IEEE, 1998.

# Image Classification

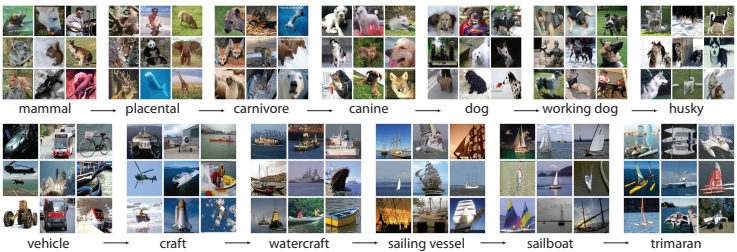



**CIFAR-10:**

- 60k color images, $32 \times 32$ pixels
- 10 classes, 6k images per class
- 50k training and 10k test images

Krizhevsky: Learning Multiple Layers of Features from Tiny Images. Technical report, 2009.

# Image Classification

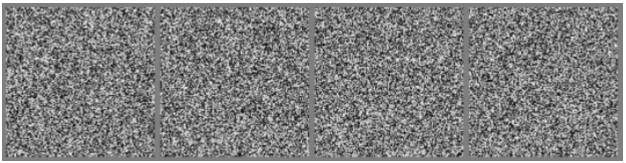


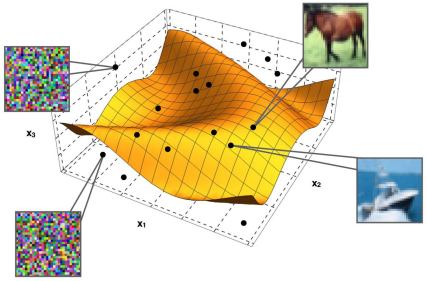

**ImageNet** (ILSVRC 2012), the benchmark of the deep learning breakthrough:

- 1000 classes, many of them fine-grained (e.g., dog breeds)
- 1.2 million training images, 50k validation images
- Color photos of varying size, typically resized to $224 \times 224$

Deng et al.: ImageNet: A Large-Scale Hierarchical Image Database. CVPR, 2009. Russakovsky et al.: ImageNet Large Scale Visual Recognition Challenge. IJCV, 2015.

# Image Classification






**Curse of Dimensionality:**

- There exist $2^{784} = 10^{236}$ possible binary images of resolution 28 x 28 pixels
- MNIST is gray-scale, thus $256^{784}$ combinations $\Rightarrow$ impossible to enumerate
- Why is image classification with just 60k labeled training images even possible?
- Answer: Images concentrated on low-dimensional manifold in high-dim. space

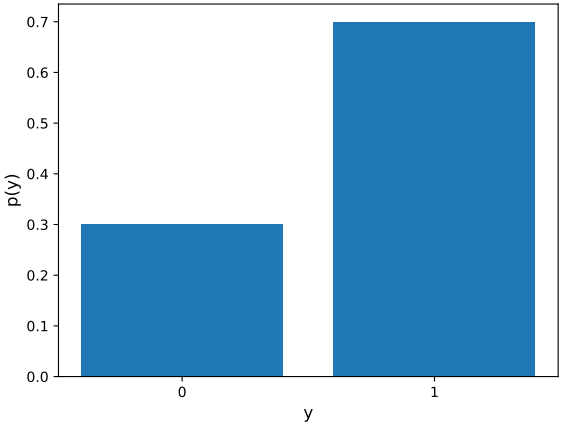
Goldt, Mézard, Krzakala and Zdeborová: Modelling the influence of data structure on learning in neural networks: the hidden manifold model. Arxiv, 2019.

# Bernoulli Distribution

**Bernoulli distribution:**

$$p(y) = \mu^y \, (1-\mu)^{(1-y)}$$

- $\mu$: probability for $y = 1$
- Handles only two classes
  e.g. (“cats” vs. “dogs”)



# Bernoulli Distribution / BCE Loss

Let $p_{model}(y|\mathbf{x}, \mathbf{w}) = f_{\mathbf{w}}(\mathbf{x})^y \left(1 - f_{\mathbf{w}}(\mathbf{x})\right)^{(1-y)}$ be a **Bernoulli distribution**. We obtain:

$$\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log \left[ f_{\mathbf{w}}(\mathbf{x}_i)^{y_i} \left(1 - f_{\mathbf{w}}(\mathbf{x}_i)\right)^{(1-y_i)} \right] \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \underbrace{-y_i \log f_{\mathbf{w}}(\mathbf{x}_i) - (1 - y_i) \log(1 - f_{\mathbf{w}}(\mathbf{x}_i))}_{\text{BCE Loss}}
\end{aligned}$$

In other words, we minimize the **binary cross-entropy (BCE)** loss.

Remark: Last layer of $f_{\mathbf{w}}(\mathbf{x})$ can be a sigmoid function such that $f_{\mathbf{w}}(\mathbf{x})^y \in [0, 1]$.

How can we scale this to multiple classes?

# Categorical Distribution

**Categorical distribution:**

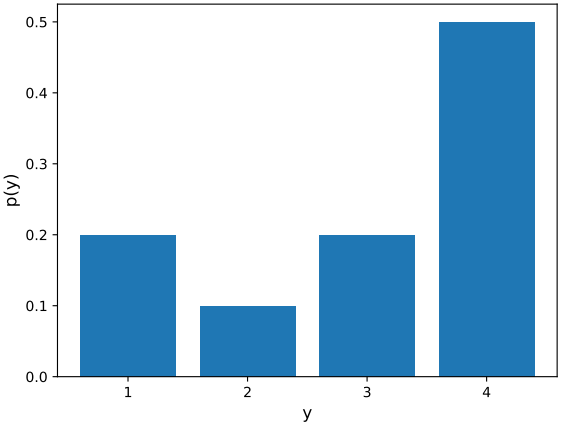
$$p(y = c) = \mu_c$$

- $\mu_c$: probability for class $c$
- Multiple classes, multiple modes

**Alternative notation:**

$$p(\mathbf{y}) = \prod_{c=1}^{C} \mu_c^{y_c}$$

- $\mathbf{y}$: "one-hot" vector with $y_c \in \{0, 1\}$
- $\mathbf{y} = (0, \ldots, 0, 1, 0, \ldots, 0)^\top$ with all zeros except for one (the true class)



# One-Hot Vector Representation

| class | $y$ | $\mathbf{y}$ |
|---|---|---|
|  | 1 | $(1,0,0,0)^\top$ |
|  | 2 | $(0,1,0,0)^\top$ |
|  | 3 | $(0,0,1,0)^\top$ |
|  | 4 | $(0,0,0,1)^\top$ |

- One-hot vector $\mathbf{y}$ with binary elements $y_c \in \{0, 1\}$
- Index $c$ with $y_c = 1$ determines the correct class, and $y_k = 0$ for $k \neq c$
- Interpretation as discrete distribution with all probability mass at the true class
- Often used in ML as it can make formalism more convenient

# Categorical Distribution / CE Loss

Let $p_{model}(\mathbf{y}|\mathbf{x}, \mathbf{w}) = \prod_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x})^{y_c}$ be a **Categorical distribution**. We obtain:

$$\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \operatorname*{argmax}_{\mathbf{w}} \sum_{i=1}^{N} \log p_{model}(\mathbf{y}_i|\mathbf{x}_i, \mathbf{w}) \\
&= \operatorname*{argmax}_{\mathbf{w}} \sum_{i=1}^{N} \log \prod_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x}_i)^{y_{i,c}} \\
&= \operatorname*{argmin}_{\mathbf{w}} \sum_{i=1}^{N} \underbrace{\sum_{c=1}^{C} -y_{i,c} \log f_{\mathbf{w}}^{(c)}(\mathbf{x}_i)}_{\text{CE Loss}}
\end{aligned}$$

In other words, we minimize the **cross-entropy (CE)** loss.

The target $\mathbf{y} = (0, \ldots, 0, 1, 0, \ldots, 0)^\top$ is a "one-hot" vector with $y_c$ its $c$'th element.

# Why the Cross-Entropy Loss?

**Cross-entropy** between the data distribution $p$ and the model $q$:

$$H(p, q) = -\sum_x p(x) \log q(x) = H(p) + \mathrm{KL}(p \,\|\, q)$$

For a one-hot target $\mathbf{y}$, this is our CE loss $-\log q(y)$. Other motivations:

- **Compression:** with the model $q$, we can encode $x$ with $-\log_2 q(x)$ bits. Minimizing $H(p, q)$ compresses the data as well as possible (at best: $H(p)$)
- **Honest probabilities:** the expected loss is minimal for $q = p$
- … and **many other motivations!**

# Softmax

How can we ensure that $f_{\mathbf{w}}^{(c)}(\mathbf{x})$ predicts a **valid Categorical (discrete) distribution?**

- We must guarantee (1) $f_{\mathbf{w}}^{(c)}(\mathbf{x}) \in [0, 1]$ and (2) $\sum_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x}) = 1$
- An element-wise sigmoid as output function would ensure (1) but not (2)
- Solution: The **softmax function** guarantees both (1) and (2):

$$\text{softmax}(\mathbf{x}) = \left( \frac{\exp(x_1)}{\sum_{k=1}^{C} \exp(x_k)}, \dots, \frac{\exp(x_C)}{\sum_{k=1}^{C} \exp(x_k)} \right)$$

- Let $\mathbf{s}$ denote the network output after the last affine layer (=scores). Then:

$$f_{\mathbf{w}}^{(c)}(\mathbf{x}) = \frac{\exp(s_c)}{\sum_{k=1}^{C} \exp(s_k)} \quad \Rightarrow \quad \log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k)$$

- Remark: $s_c$ is a direct contribution to the loss function, i.e., it does not saturate

# Softmax

- Predicting $C$ values/scores overparameterizes the Categorical distribution
- As the distribution sums to 1 only $C1$ parameters are necessary
- Example: Consider $C = 2$ and fix one degree of freedom ($x_2 = 0$):

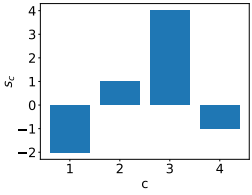
$$\begin{aligned}
\text{softmax}(\mathbf{x}) &= \left( \frac{\exp(x_1)}{\exp(x_1) + \exp(x_2)}, \frac{\exp(x_2)}{\exp(x_1) + \exp(x_2)} \right) \\
&= \left( \frac{\exp(x_1)}{\exp(x_1) + 1}, \frac{1}{\exp(x_1) + 1} \right) \\
&= \left( \frac{1}{1 + \exp(-x_1)}, 1 - \frac{1}{1 + \exp(-x_1)} \right) \\
&= (\sigma(x_1), 1 - \sigma(x_1))
\end{aligned}$$

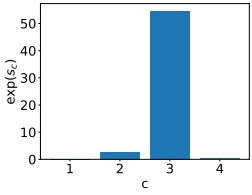
- The softmax is a **multi-class generalization** of the sigmoid function
- In practice, the overparameterized version is often used (simpler to implement)

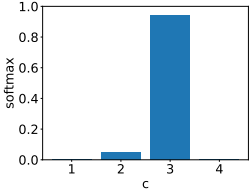
# Softmax

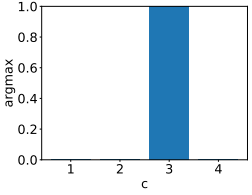
$$\text{softmax}(\mathbf{s}) = \left( \frac{\exp(s_1)}{\sum_{k=1}^{C} \exp(s_k)}, \ldots, \frac{\exp(s_C)}{\sum_{k=1}^{C} \exp(s_k)} \right)$$

- The name “softmax” is confusing, “soft argmax” would be more precise as it is a continuous and differentiable version of argmax (in one-hot representation)
- Example with four classes:




https://jramkiss.github.io/2020/07/29/overconfident-nn/

# Softmax

- Softmax responds to differences between inputs
- It is invariant to adding the same scalar to all its inputs:

$$\mathrm{softmax}(\mathbf{x}) = \mathrm{softmax}(\mathbf{x} + c)$$

- We can therefore derive a numerically more stable variant:

$$\mathrm{softmax}(\mathbf{x}) = \mathrm{softmax}(\mathbf{x} - \max_{k=1..L} x_k)$$

- Allows accurate computation even when $\mathbf{x}$ is large
- Illustrates again that softmax depends on differences between scores

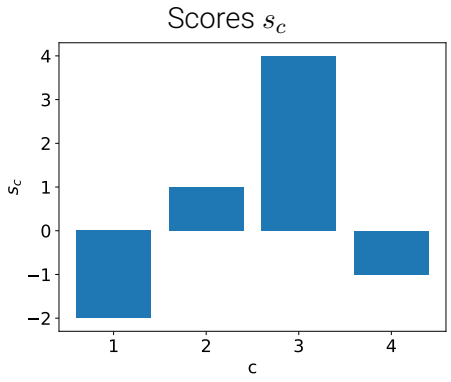
# Log Softmax

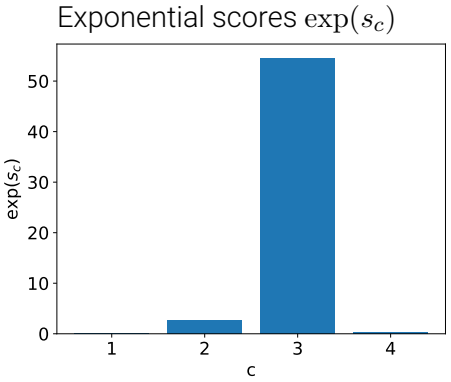
**Intuition:** Assume $c$ is the correct class. Our goal is to maximize the log softmax:

$$\log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k)$$

- The first term encourages the score $s_c$ for the correct class $c$ to increase
- The second term encourages all scores in $\mathbf{s}$ to decrease
- The second term can be approximated by: $\log \sum_{k=1}^{C} \exp(s_k) \approx \max_k s_k$ as $\exp(s_k)$ is insignificant for all $s_k < \max_k s_k$
- Thus, the loss always strongly penalizes the most active incorrect prediction
- If the correct class already has the largest score (i.e., $s_c = \max_k s_k$), both terms roughly cancel and the example will contribute little to the overall training cost

# Log Softmax Example




- The second term becomes: $\log \sum_{k=1}^{C} \exp(s_k) = 4.06 \approx s_3 = \max_k s_k$
- For $c = 2$ we obtain: $\log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k) = 1 - 4.06 \approx -3$
- For $c = 3$ we obtain: $\log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k) = 4 - 4.06 \approx 0$

# Cross Entropy Loss with Softmax Example

**Putting it together:** Cross Entropy Loss for a single training sample $(\mathbf{x}, \mathbf{y}) \in \mathcal{X}$:

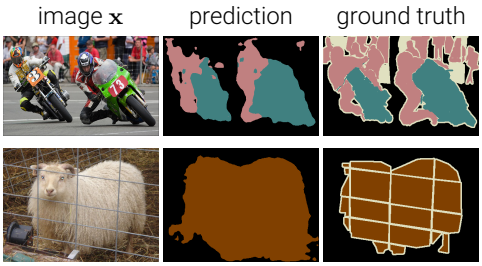
$$\text{CE Loss: } \sum_{c=1}^{C} -y_c \log f_{\mathbf{w}}^{(c)}(\mathbf{x})$$

Example: Suppose $C = 4$ and $4$ samples $\mathbf{x}$ with labels $\mathbf{y}$ (log = natural logarithm)

| Input $\mathbf{x}$ | Label $\mathbf{y}$ | Predicted scores $\mathbf{s}$ | softmax$(\mathbf{s}) = \mathbf{f}_{\mathbf{w}}(\mathbf{x})$ | CE Loss |
|---|---|---|---|---|
|  | $(1,0,0,0)^\top$ | $(+3,+1,-1,-1)^\top$ | $(0.85,0.12,0.02,0.02)^\top$ | 0.16 |
|  | $(0,1,0,0)^\top$ | $(+3,+3,+1,+0)^\top$ | $(0.46,0.46,0.06,0.02)^\top$ | 0.78 |
|  | $(0,0,1,0)^\top$ | $(+1,+1,+1,+1)^\top$ | $(0.25,0.25,0.25,0.25)^\top$ | 1.38 |
|  | $(0,0,0,1)^\top$ | $(+3,+2,+3,-1)^\top$ | $(0.42,0.16,0.42,0.01)^\top$ | 4.87 |

- Sample 4 contributes most strongly to the loss function! (elephant in the room)

# Example: Semantic Segmentation



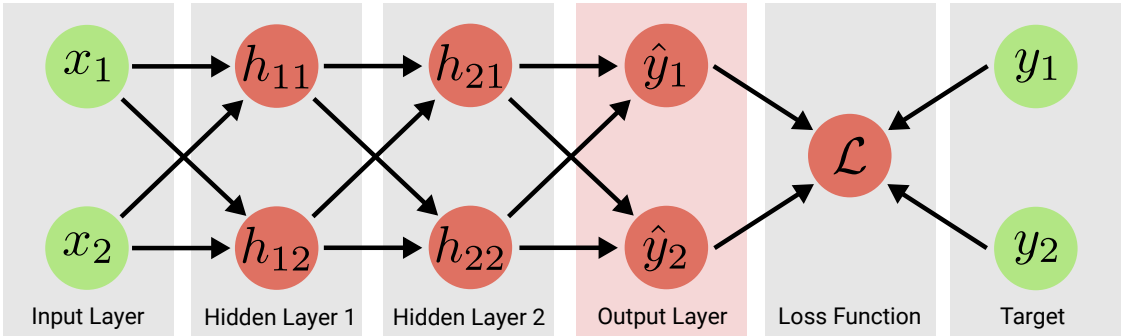

- **Classify every pixel** into $C$ classes
- Output: softmax over $C$ classes per pixel
- Loss: **cross-entropy per pixel**, summed

$$\mathcal{L} = \sum_{p} \sum_{c=1}^{C} -y_{p,c} \log \hat{y}_{p,c}$$

- Like pixel-wise $L_2$: one term per pixel

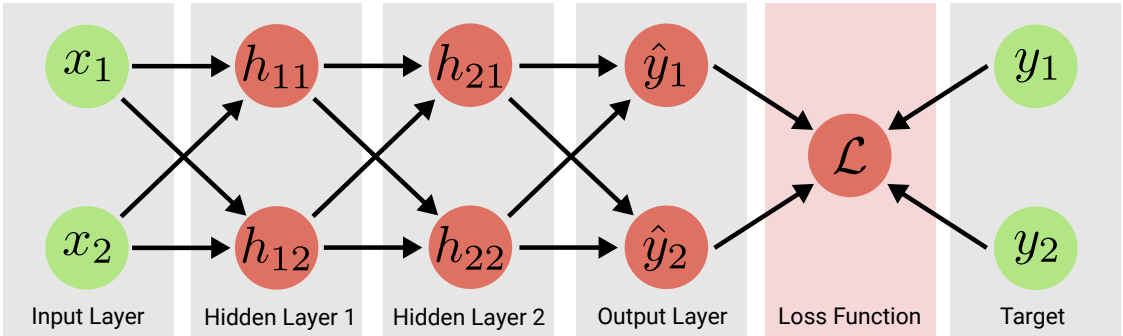
Long, Shelhamer and Darrell: Fully Convolutional Networks for Semantic Segmentation. CVPR, 2015.

# Output Layer for Classification Problems



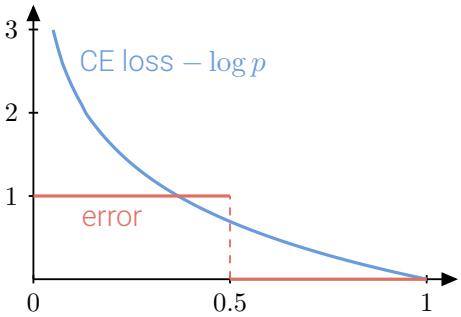

- For 2 classes, we can predict 1 value and use a sigmoid, or 2 values with softmax
- For $C > 2$ classes we typically predict $C$ scores and use a softmax non-linearity

# Loss Function for Classification Problems




- For $2$ classes, we use the binary cross-entropy loss (BCE)
- For $C > 2$ classes, we use the cross-entropy loss (CE)

# The Loss is a Proxy




Two classes: correct if $p > 0.5$

- In training, we minimize the **loss**, e.g. the cross-entropy (CE)
- We often care about a different **metric**, e.g. the classification error
- The error is **piecewise constant**: its gradient is zero almost everywhere
- The CE loss is a differentiable **proxy**: lowering it pushes $p$ towards 1

# What Do We Actually Care About?

The loss is a tool for training. What we care about are properties of the model:

- **Generalization:** a low error on new data, not only on the training data
- **Cost of errors:** missing a tumor is worse than a false alarm
- **Calibration:** predicted probabilities that we can trust
- **Robustness:** small changes of the input do not change the prediction
- **Fairness and safety:** similar errors across groups, no harmful outputs

**Example:** language models are trained to predict the next token with the CE loss, but we want them to be helpful, correct and safe

# Stretching Break

5 Minutes

# 4.2
# Preprocessing and Initialization

# Recap: Training with Stochastic Gradient Descent

**Algorithm** for training an MLP using (stochastic) gradient descent:

1. Initialize weights $\mathbf{w}$, pick learning rate $\eta$ and minibatch size $|\mathcal{X}_{\text{batch}}|$
2. Draw (random) minibatch $\mathcal{X}_{\text{batch}} \subseteq \mathcal{X}$
3. For all elements $(\mathbf{x}, \mathbf{y}) \in \mathcal{X}_{\text{batch}}$ of minibatch (in parallel) do:
   3.1 Forward propagate $\mathbf{x}$ through network to calculate $\mathbf{h}_1, \mathbf{h}_2, \ldots, \hat{\mathbf{y}}$
   3.2 Backpropagate gradients through network to obtain $\nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$
4. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{|\mathcal{X}_{\text{batch}}|} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{X}_{\text{batch}}} \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$
5. If validation error decreases, go to step 2, otherwise stop

# Training Stability

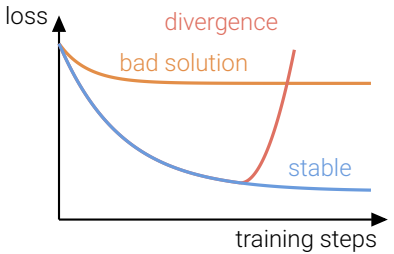
# Training Neural Networks in Practice

**Goal:** low validation loss after optimization

**Prerequisite:** a **stable** optimization process

- The model improves with every gradient step
- No convergence to a bad solution
  (e.g., the same output for all inputs)
- No divergence (loss explodes, `NaN`)

**This section:** a stable start through
input normalization and weight initialization



# Training Stability

Consider a network with layers $l = 1, \ldots, L$, activations $\mathbf{h}_l$ and weights $\mathbf{A}_l$.

**Requirement:** During the entire training process, all quantities are **of order one**, $\Theta(1)$:

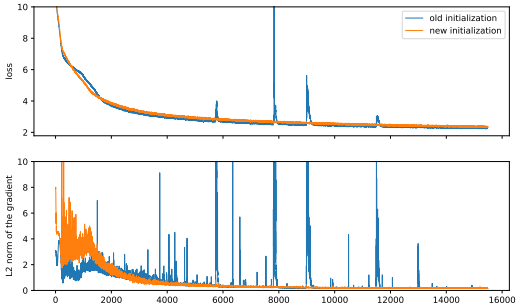
- the **activations** $\mathbf{h}_l$ (forward pass)
- the **gradients** $\partial \mathcal{L} / \partial \mathbf{h}_l$ and $\partial \mathcal{L} / \partial \mathbf{A}_l$ (backward pass)
- the **updates** $\Delta \mathbf{A}_l$ and their effect $\Delta \mathbf{h}_l$ on the next forward pass[1]

$\Theta(1)$: neither very small nor very large, independent of depth, width and training step

- **Vanishing** ($\to 0$): all inputs get the same representation, no gradient, no learning
- **Exploding** ($\to \infty$): loss spikes, divergence, overflow in float32 / bfloat16 (`NaN`)
- In practice, we monitor the norms of these quantities during training

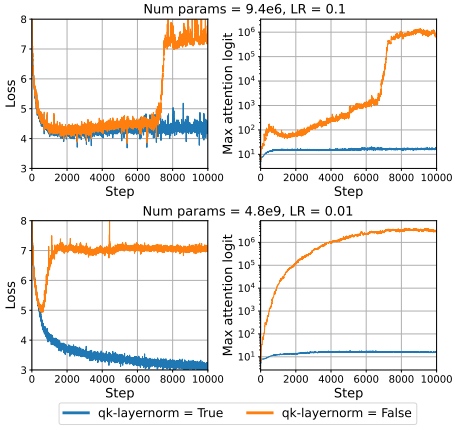
[1] Reasoning about the updates is an advanced topic. We focus on the activations and gradients.

# Training Instabilities in Practice: Loss Spikes




- Two training runs of a language model that differ only in the **weight initialization**
- Old initialization: the loss and the gradient norm **spike** repeatedly
- New initialization: stable training

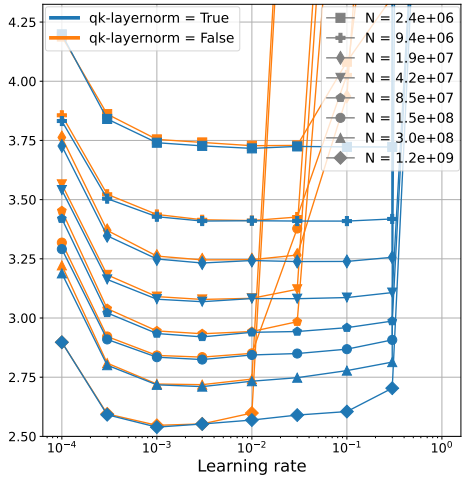
# Training Instabilities in Practice: Divergence




- Transformers with 9.4M and 4.8B parameters, trained with a large learning rate
- The **attention logits**, an intermediate quantity of the transformer, grow to $10^6$
- Shortly after, the **loss diverges**
- Normalizing the inputs of this operation (qk-layernorm) keeps them $\Theta(1)$: stable

Wortsman et al.: Small-scale proxies for large-scale Transformer training instabilities. ICLR, 2024.

# Training Instabilities in Practice: Learning Rate Sensitivity




- Final loss as a function of the learning rate, transformers with 2.4M to 1.2B parameters
- Too small: slow progress. Too large: **divergence**
- The best learning rate lies close to the edge of divergence
- Stabilization (here: qk-layernorm) widens the range of good learning rates
- The sensitivity to the learning rate grows with the model size

Wortsman et al.: Small-scale proxies for large-scale Transformer training instabilities. ICLR, 2024.

# Input Normalization

# Input Normalization

The input $\mathbf{x}$ is the first activation of the network, so it should be $\Theta(1)$ as well.

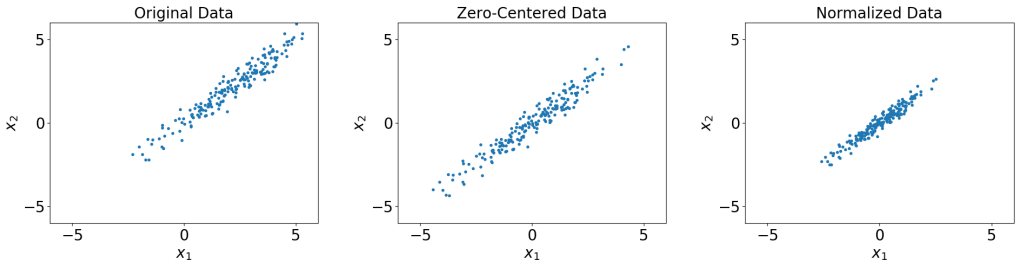
**Raw inputs usually are not:**

- Images: pixel values in $\{0, \ldots, 255\}$
- Tabular data: features in different units (age in years, income in euros)
- Audio and sensor data: arbitrary physical scales

**Two goals:**

- **Scale:** every input feature has a standard deviation of about $1$
- **Zero-centering:** every input feature has mean $0$ (next slide)

# Input Normalization




- **Zero-center:** $x_{i,j} \leftarrow x_{i,j} - \mu_j$ with $\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$
- **Normalization:** $x_{i,j} \leftarrow x_{i,j}/\sigma_j$ with $\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$

# Input Normalization: Why Zero-Centering and Scaling?

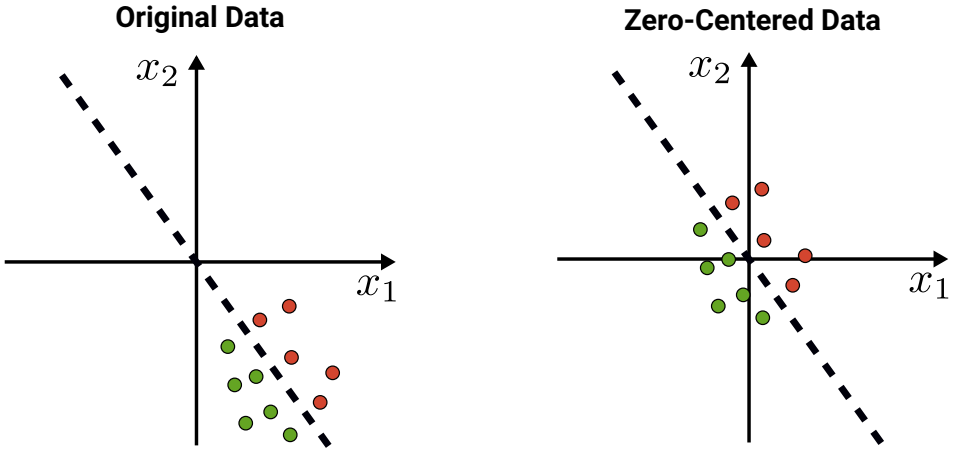
Take one linear unit with squared loss, $\mathcal{L}(\mathbf{a}) = \frac{1}{2}\,\mathbb{E}\big[(\mathbf{a}^\top \mathbf{x} - y)^2\big]$. Then

$$\nabla \mathcal{L} = \mathbb{E}\big[(\mathbf{a}^\top \mathbf{x} - y)\,\mathbf{x}\big], \qquad H = \mathbb{E}[\mathbf{x}\mathbf{x}^\top] = \boldsymbol{\Sigma} + \boldsymbol{\mu}\boldsymbol{\mu}^\top$$

- The curvature of the loss is the **second moment of the inputs**
- Gradient descent needs $\eta < 2/\lambda_{\max}$ and is slow along directions with small $\lambda_i$
- **Mean** $\boldsymbol{\mu} \neq 0$: extra eigenvalue $\geq \|\boldsymbol{\mu}\|^2$ in direction $\boldsymbol{\mu} \Rightarrow$ small $\eta$, slow zig-zag
- **Zero-centering** removes it; **unit variance** evens out the curvature

LeCun, Bottou, Orr and Müller: Efficient BackProp. Neural Networks: Tricks of the Trade, 1998.

# Input Normalization




- Classification loss becomes less sensitive to changes in the weight matrix

# Input Normalization in Practice

**Common Practices for Images:**

- AlexNet: Subtract mean image
  (mean image: $W \times H \times 3$ numbers)
- VGGNet: Subtract per-channel mean
  (mean along each channel: $3$ numbers)
- ResNet: Subtract per-channel mean and divide by per-channel std. dev.
  (mean along each channel: $3$ numbers)

**Language Models:**

- No preprocessing: the inputs are learned token embeddings

# Weight Initialization

# Weight Initialization

**Question:** How to best initialize the weights $\mathbf{w}$?

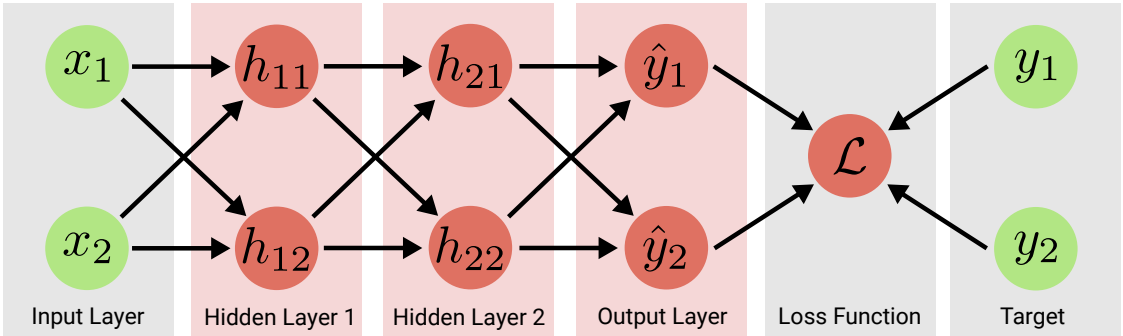
Consider a layer in a Multi-Layer Perceptron:

$$g(x) = g\left(\sum_i a_i x_i + b\right)$$

The gradient wrt. parameter $a_i$ is given by:

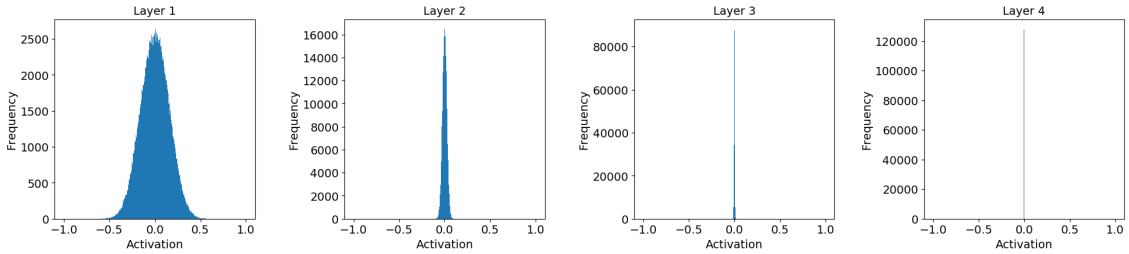
$$\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x}x_i$$

# First Attempt: Constant Initialization




- Simple solution: set all parameters to a constant, e.g., $a_i = 0$ and $b = 0$
- All units of a layer compute the same output $g\left(\sum_i a_i x_i + b\right)$ and receive the same gradient $\partial \mathcal{L} / \partial a_i$
- The units stay identical during training $\Rightarrow$ learning is not possible

# Second Attempt: Small Random Numbers

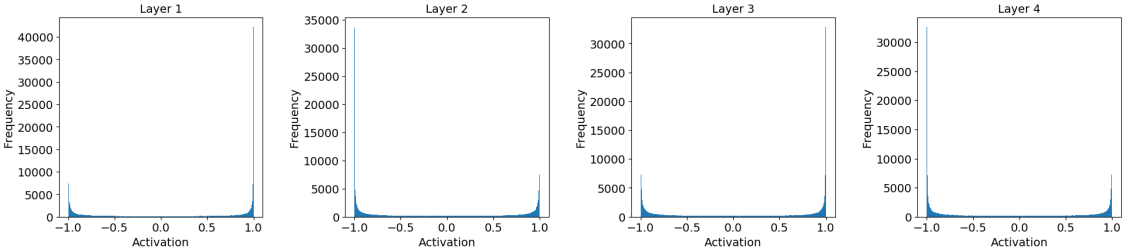



**Tanh Activation Function:**

- Draw weights independently from Gaussian with small std. dev ($\sigma = 0.01$)
- Activations (=activation function outputs) in deeper layers tend towards zero
- Gradients wrt. weights thus also tend towards zero $\Rightarrow$ no learning:

$$\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x}x_i = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x}0 = 0$$

# Third Attempt: Large Random Numbers




**Tanh Activation Function:**

- Draw weights independently from Gaussian with large std. dev ($\sigma = 0.2$)
- All activation functions saturate
- Local gradients are all becoming zero $\Rightarrow$ no learning:

$$\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x}x_i = \frac{\partial \mathcal{L}}{\partial g}\,0\,x_i = 0$$

# Choosing the Right Initialization Scale

Consider one unit of a layer with $D_{in}$ inputs (before the activation function):

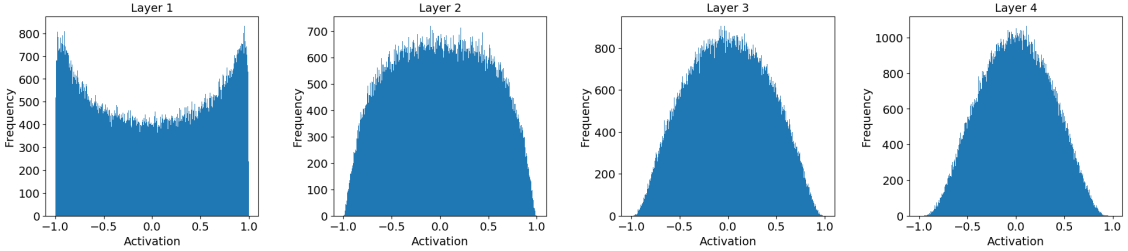
$$h = \sum_{i=1}^{D_{in}} a_i \, x_i$$

- Assume the inputs $x_i$ are zero-mean and $\Theta(1)$ (normalized inputs or activations)
- Draw the weights $a_i$ independently with zero mean and standard deviation $\sigma$
- Independent terms: the sum grows like $\sqrt{D_{in}}$ (central limit theorem):

$$\mathrm{Var}(h) = \sum_{i=1}^{D_{in}} \mathrm{Var}(a_i \, x_i) = D_{in} \, \sigma^2 \, \mathrm{Var}(x_i) \qquad \Rightarrow \qquad |h| \approx \sqrt{D_{in}} \, \sigma$$

- With $\sigma^2 = 1/D_{in}$ we get $\mathrm{Var}(h) = \mathrm{Var}(x_i)$: the scale is preserved, independent of the width $D_{in}$

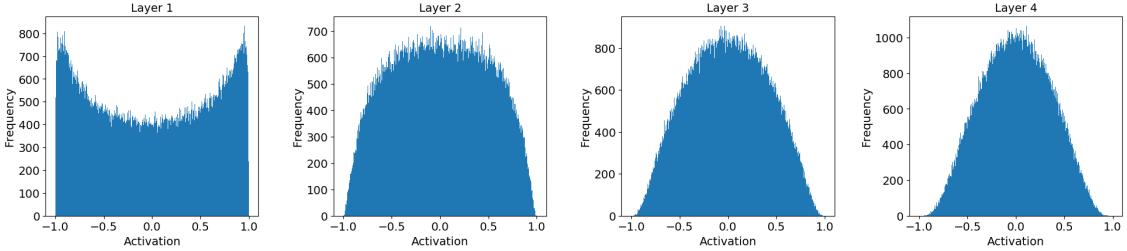
# Principled Initialization for Tanh Activation




**Tanh Activation Function:**

- Draw weights independently from Gaussian with std. dev $\sigma = 1/\sqrt{D_{in}}$
- $D_{in}$ denotes the dimension of the input to each layer
- Activation distribution now well scaled across all layers

# Glorot or Xavier Initialization

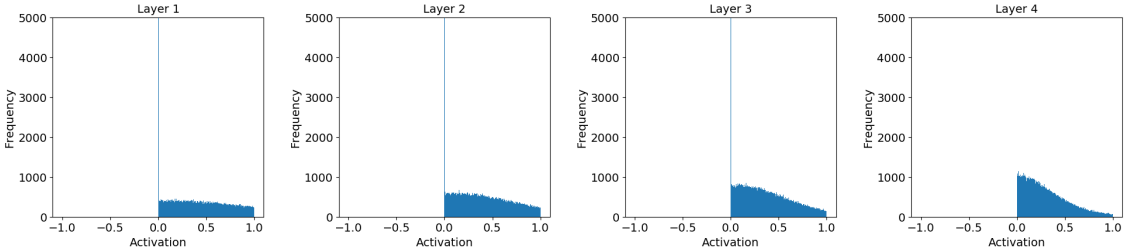



**Tanh Activation Function:**

- Glorot et al. draw weights independently from $U\left(-\sqrt{\frac{6}{D_{in}+D_{out}}}, \sqrt{\frac{6}{D_{in}+D_{out}}}\right)$
- $D_{in}/D_{out}$ denote the dimension of the input to/output of each layer
- Variance $2/(D_{in}+D_{out})$: a compromise between the forward pass $(1/D_{in})$ and the backward pass $(1/D_{out})$

Glorot and Bengio: Understanding the difficulty of training deep feedforward neural networks. AISTAT, 2010.

# Initialization for ReLU Activation

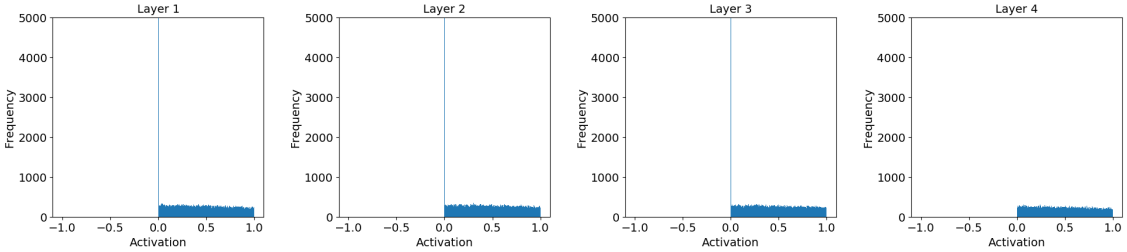



**ReLU Activation Function:**

- Xavier initialization assumes zero centered activation function
- For ReLU, activations again start collapsing to zero for deeper layers

Glorot and Bengio: Understanding the difficulty of training deep feedforward neural networks. AISTAT, 2010.

# He Initialization for ReLU Activation




**ReLU Activation Function:**

- Since ReLU is restricted to positive outputs, variance must be doubled
- He et al. draw weights independently from Gaussian with $\sigma^2 = 2/D_{in}$
- Activation distribution now well scaled across all layers

He, Zhang, Ren and Sun: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. ICCV, 2015.

# Initialization in Practice

- **Biases:** initialized to zero – the random weights already break the symmetry
- **Distribution:** Gaussian or uniform with the chosen variance, both are common ($U(-a, a)$ has variance $a^2/3$: this is where the $6$ in Glorot's bound comes from)
- **PyTorch default** (`nn.Linear`, `nn.Conv2d`): weights from $U\big(-1/\sqrt{D_{in}},\ 1/\sqrt{D_{in}}\big)$, i.e., variance $1/(3D_{in})$; the biases are drawn from the same distribution, not zero
- **Common pattern:** $\sigma^2 = c/D_{in}$ with $c = 1/3$ (PyTorch), $1$ (Xavier), $2$ (He), or with $D_{in}$ replaced by $(D_{in} + D_{out})/2$ (Glorot)

# Initialization in Modern Networks

- **The defaults are usually fine:** in PyTorch, one rarely writes initialization code
- **Normalization layers** (later in the course) rescale the activations after every layer $\Rightarrow$ training becomes much less sensitive to the initialization
- **The optimizer helps, too:** adaptive methods such as Adam (later in the course) rescale the updates per parameter, so badly scaled gradients hurt less
- **But careful initialization remains important**, especially as models are scaled up: normalization makes the activations independent of the weight scale $\sigma$, but the gradients wrt. the weights scale like $1/\sigma$ – the initialization still sets the step size

# Implementation

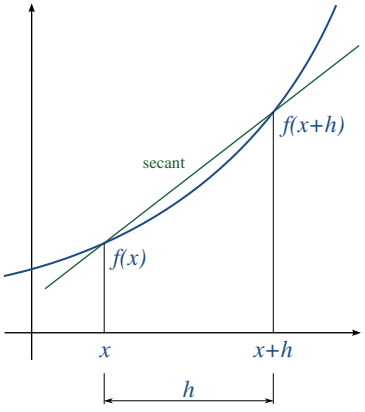
# Numerical Differentiation

- When implementing the backward pass of activation, output or loss functions it is important to ensure that all gradients are correct!
- Verify via Newton's difference quotient:

$$\frac{\partial f(x)}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- Even better: Symmetric difference quotient

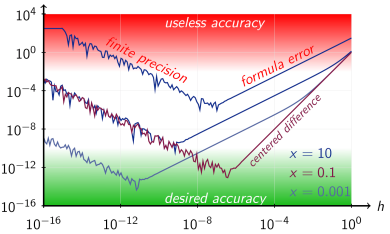
$$\frac{\partial f(x)}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$



# Numerical Differentiation

How to choose $h$?

- For $h = 0$ the expression is undefined
- Choose $h$ to trade-off:
  - Rounding error (finite precision)
  - Approximation error (wrong)
- Good choice: $\sqrt[3]{\epsilon}$ with $\epsilon$ the machine precision
- Examples:
  - $\epsilon = 6 \times 10^{-8}$ for single precision (32 bit)
  - $\epsilon = 1 \times 10^{-16}$ for double precision (64 bit)

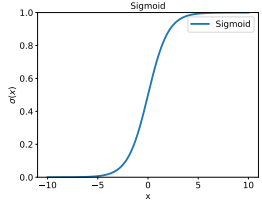



en.wikipedia.org/wiki/Numerical_differentiation

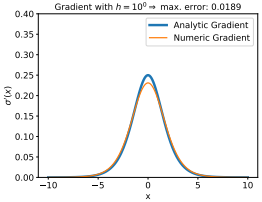
# Numerical Differentiation

Example: Sigmoid derivative using symmetric differences with single precision:

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

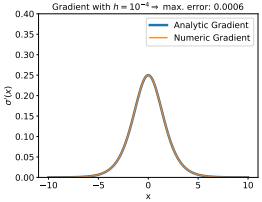



Sigmoid



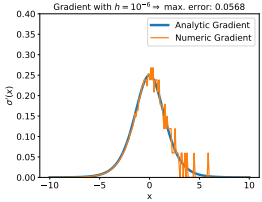

$h = 10^0$

max. error $0.0189$




$h = 10^{-4}$

max. error $0.0006$

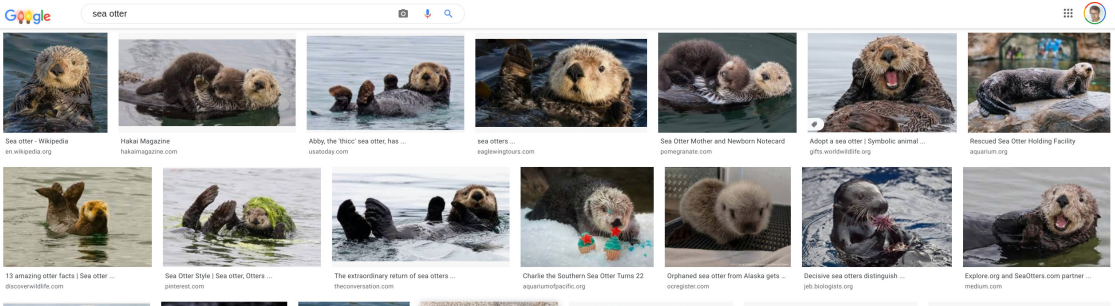



$h = 10^{-6}$

max. error $0.0568$

# 4.3
# Data Augmentation

# Data Augmentation

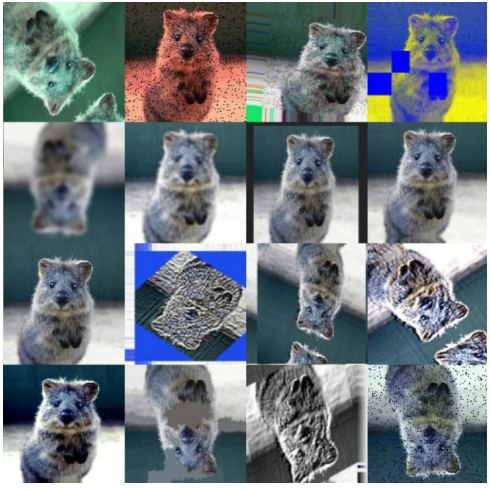



**Motivation:**

- Deep neural networks must be **invariant** to a wide variety of input variations
- Often **large intra-class variation** in terms of pose, appearance, lighting, etc.

# Data Augmentation

- Best way towards better generalization is to **train on more data**
- However, data in practice often limited
- Goal of data augmentation: create **“fake” data** from the existing data (on the fly) and add it to the training set
- New data must **preserve semantics**
- Even **simple operations** like translation or adding per-pixel noise often already greatly improve generalization
- `https://github.com/aleju/imgaug`



# Augmenting Images

# Geometric Transformations

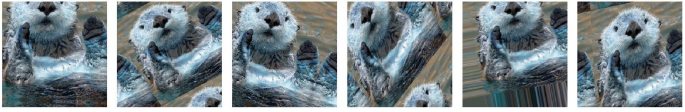
Crop



Flip



Affine



- Crop, flip, scale, rotate, shear: the object stays the same object
- Random crops and flips: the standard minimum for classification

# Photometric Transformations and Noise

Color



Noise

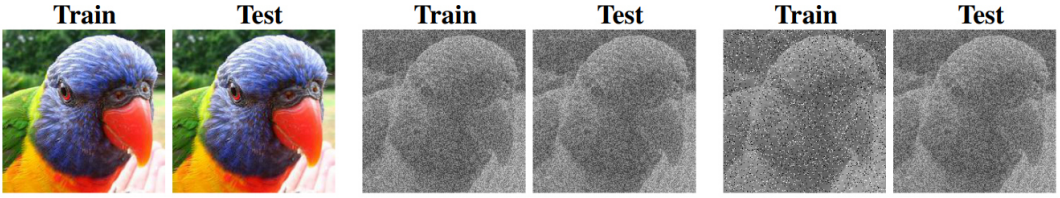


Cutout



- Color and blur: lighting and camera change, not the object
- Noise and cutout: no reliance on single pixels or patches

# Why Add Noise? Humans vs. Deep Networks

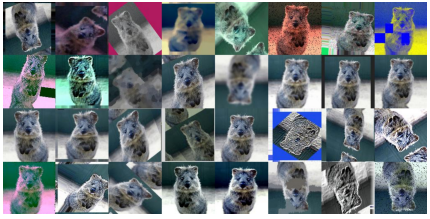



(a) Super-human performance (b) Super-human performance (c) Chance level performance

Figure 1: Classification performance of ResNet-50 trained from scratch on (potentially distorted) ImageNet images. **(a)** Classification performance when trained on standard colour images and tested on colour images is close to perfect (better than human observers). **(b)** Likewise, when trained and tested on images with additive uniform noise, performance is super-human. **(c)** Striking generalisation failure: When trained on images with salt-and-pepper noise and tested on images with uniform noise, performance is at chance level—even though both noise types do not seem much different to human observers.

# Random Policies: RandAugment

- Operations are **chained**, each with a random strength
- AutoAugment (2019): policy found by reinforcement learning, expensive search
- **RandAugment** (2020): $N$ random operations per image, strength $M$
  Two hyperparameters, no search, equally good



Cubuk, Zoph, Shlens and Le: RandAugment: Practical Automated Data Augmentation with a Reduced Search Space. CVPR Workshops, 2020.

# Mixing Inputs and Labels: Mixup and CutMix



$\mathbf{x}_i$: dog

$\mathbf{y}_i = (1, 0, 0, 0)^\top$



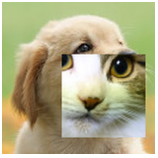
$\mathbf{x}_j$: cat

$\mathbf{y}_j = (0, 1, 0, 0)^\top$



mixup, $\lambda = 0.6$

$\tilde{\mathbf{y}} = (0.6, 0.4, 0, 0)^\top$



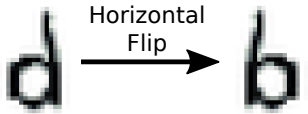
CutMix, 30% patch

$\tilde{\mathbf{y}} = (0.7, 0.3, 0, 0)^\top$

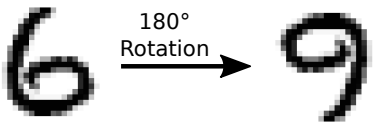
- **Mixup:** blend two training examples, inputs and labels alike:
  $\tilde{\mathbf{x}} = \lambda\, \mathbf{x}_i + (1 - \lambda)\, \mathbf{x}_j, \quad \tilde{\mathbf{y}} = \lambda\, \mathbf{y}_i + (1 - \lambda)\, \mathbf{y}_j, \quad \lambda \sim \text{Beta}(\alpha, \alpha)$, e.g., $\alpha = 0.2$
- **CutMix:** paste a patch of $\mathbf{x}_j$ into $\mathbf{x}_i$, labels mixed by the patch area
- The target is now **soft**: a distribution over the classes instead of a single class

# Output Transformations

- For some classification tasks, e.g., **handwritten letter recognition**, be careful to not apply transformations that would change the output class
- Example 1: Horizontal flips changes the interpretation of the letter 'd':

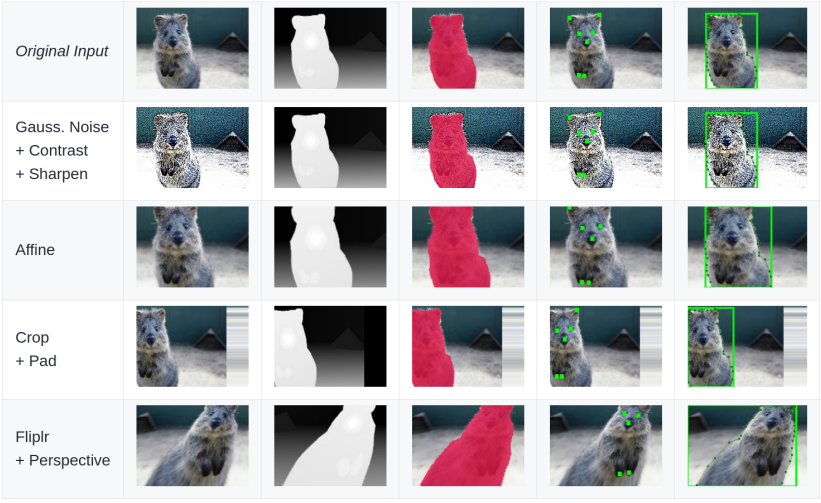



- Example 2: 180° rotations changes the interpretation of the number '6':




- Remark: For **general object recognition**, flips and rotations can often be useful!

# Output Transformations




- For dense prediction tasks (depth/instance/keypoints), also **transform targets**

# Beyond Images

# Tabular Data

| | age | income | children | default? |
|---|---|---|---|---|
| row 1 | 34 | 52,000 | 1 | no |
| row 2 | 29 | 41,000 | 0 | **yes** |
| row 3 | 45 | 38,000 | 2 | **yes** |
| SMOTE (between rows 2 and 3) | 37 | 39,500 | 1 | **yes** |

- Few known invariances: there is no “flip” of a customer
- Small **noise** on numeric features; mixup also works on rows
- Class imbalance: **SMOTE** creates minority-class rows by interpolating between a row and one of its nearest minority-class neighbours
- Much less common than for images: more real data beats fake data

Chawla, Bowyer, Hall and Kegelmeyer: SMOTE: Synthetic Minority Over-sampling Technique. JAIR, 2002.

# Text: From Word Edits to Synthetic Data

| | |
|---|---|
| **Original** (label: positive) | “The movie was surprisingly good, I would watch it again.” |
| Synonym replacement | “The **film** was surprisingly good, I would **see** it again.” |
| Random deletion | “The movie was surprisingly good, would watch again.” |
| Back-translation (via DE) | “The film was surprisingly good and I would watch it again.” |
| LLM paraphrase | “Didn’t expect much, but it turned out great.” |

- Word edits are cheap but can change the meaning (negations!)
- Today an **LLM writes the variations** (paraphrases, new examples)
- At scale: LLM-written **synthetic text** becomes LLM training data

Wei and Zou: EDA: Easy Data Augmentation Techniques. EMNLP, 2019. Sennrich, Haddow and Birch: Improving NMT with Monolingual Data. ACL, 2016.

# Break

- **Now: 30 minute break**
- **After that: Tutorial**
  - First: discussion of the **quiz**



**Quiz time!**

forms.gle/6AgcpNczkK95fNHA8