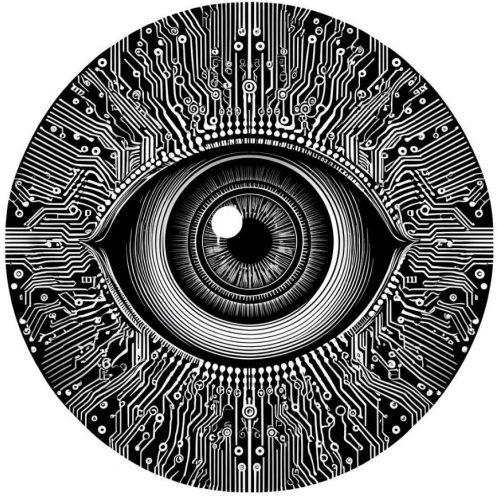
Workshop 3 - Introduction to Image Classification

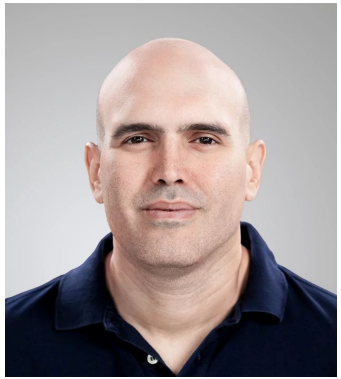


Antonio Rueda-Toicen

VOXEL51

About me

- AI Researcher at Hasso Plattner Institute, AI Engineer & DevRel for Voxel51
- Organizer of the Berlin Computer Vision Group
- Instructor at Nvidia's Deep Learning Institute and Berlin's Data Science Retreat
- Preparing a MOOC for OpenHPI (to be published in May)



LinkedIn

Agenda

- Building a feedforward network for classification in PyTorch
- Metrics for classification and experiment tracking
- PyTorch's Dataset and DataLoader

Notebooks

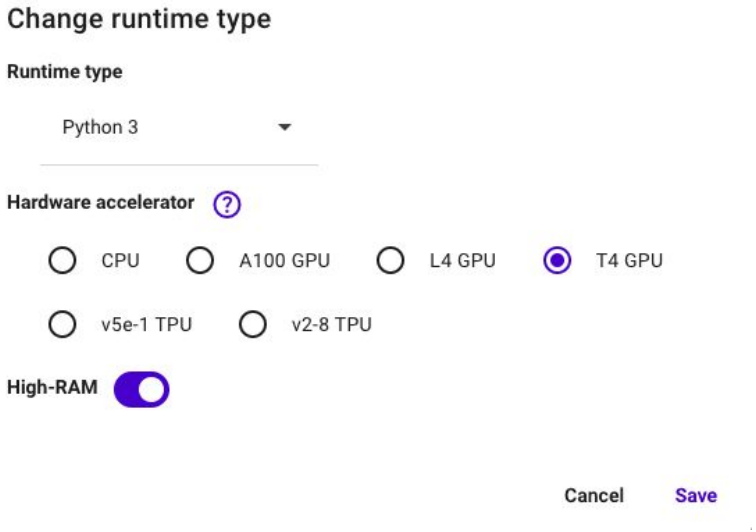
- Classifying breeds of cats and dogs
  - Colab notebook
- Digit recognition on the MNIST dataset (competition starter)
  - Kaggle notebook

Setup for today

- Setting up a Weights and Biases API token
- Setting up free GPU and TPU access in Kaggle Notebooks

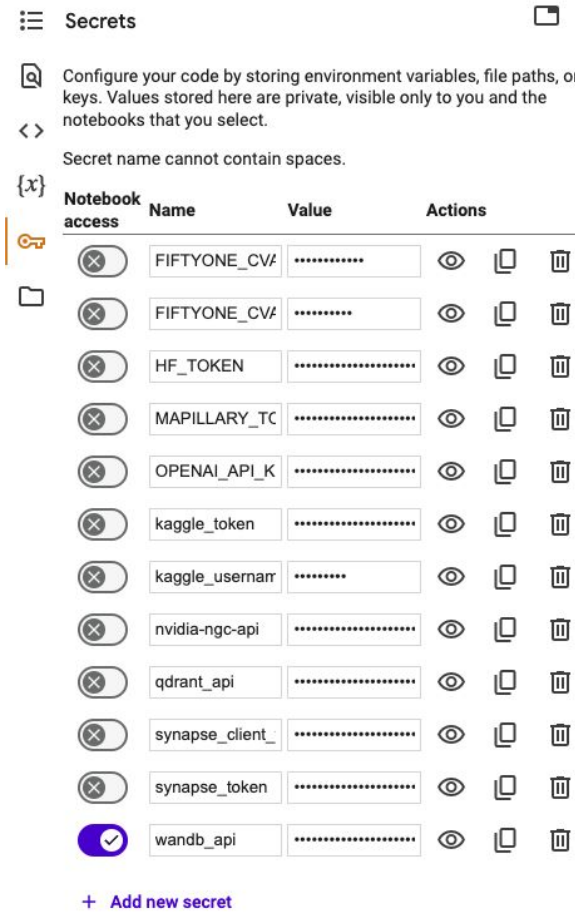
Google Colab Runtime Setup for Training

- Runtime -> Change Runtime type

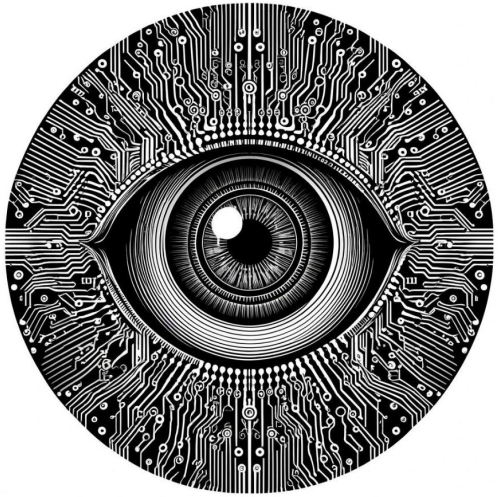


Google Colab Setup for wandb API keys

- Go to wandb.ai/authorize to find your API key
- Go to Secrets -> “Add New Secret”
  in your Colab environment



Classifying Pets According to Breed



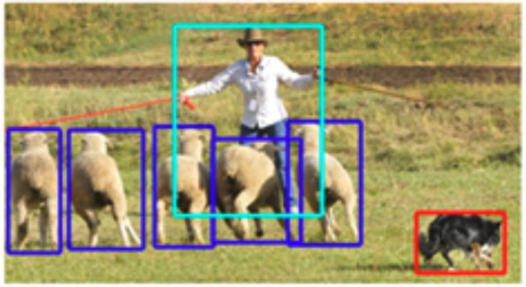
What is the breed of my dog?

The Oxford-IIIT Pets dataset

Oxford-IIIT Pet Dataset

Localized vs Non-localized Image Classification

Classification



Object Detection

Instance Segmentation

Apps that classify pet breeds

App Store Vorschau

Öffne den Mac App Store, um Apps zu kaufen und zu laden.

Dog Scanner Dog Breed Photo ID 4+

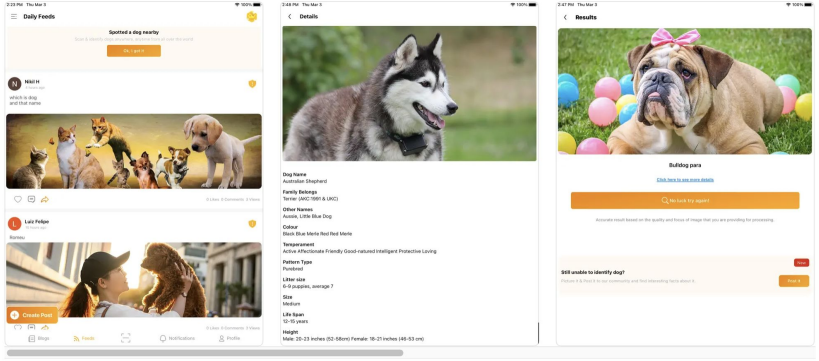
Zain ul Abdin

Entwickelt für iPad

Gratis · In-App-Käufe möglich

Anzeigen in: Mac App Store ↗

Screenshots iPad iPhone



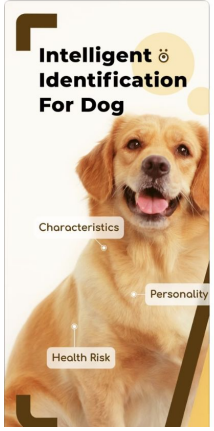
DogSnap:Dog breed scanner&Care

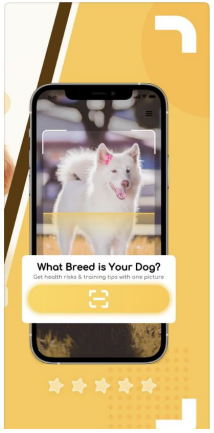
Breed Identifier,Photo Scanner

Designed for iPhone. Not verified for macOS.

Get In-App Purchases

AGE	CATEGORY	DEVELOPER	LANGUAGE	SIZE
4+	Education	HICLUB, INC.	EN	155,4
Years Old			English	MB





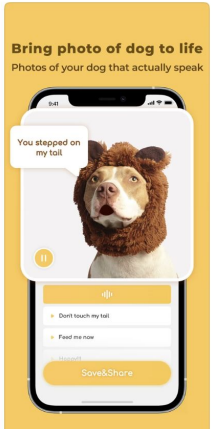
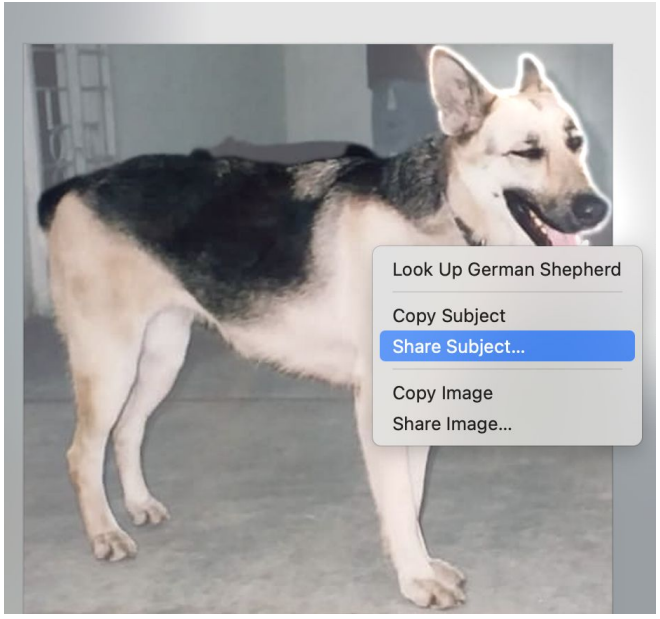
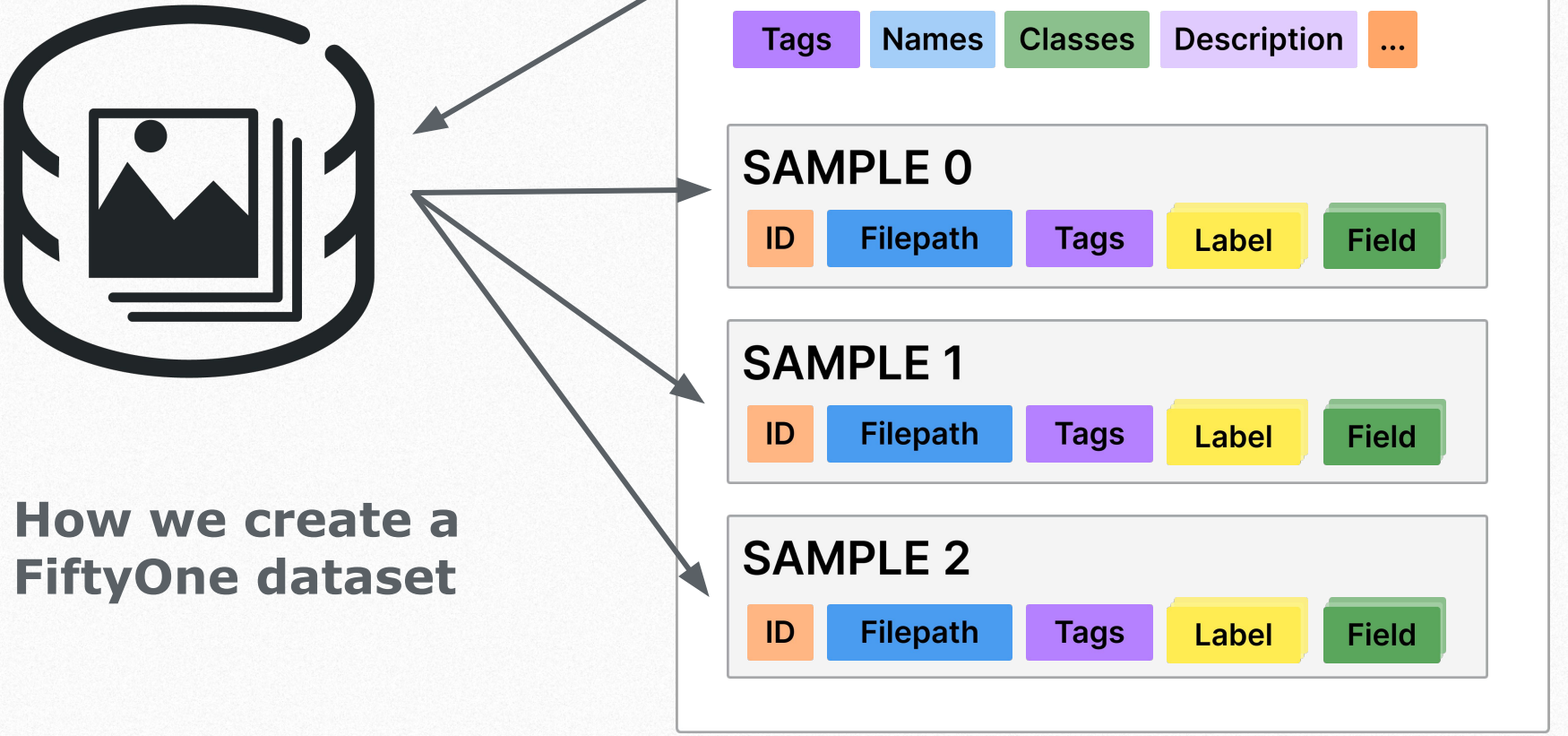


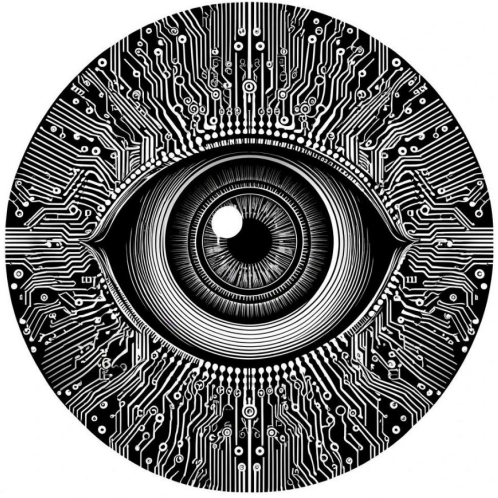
Image lens in MacOS’s Finder





How we create a FiftyOne dataset

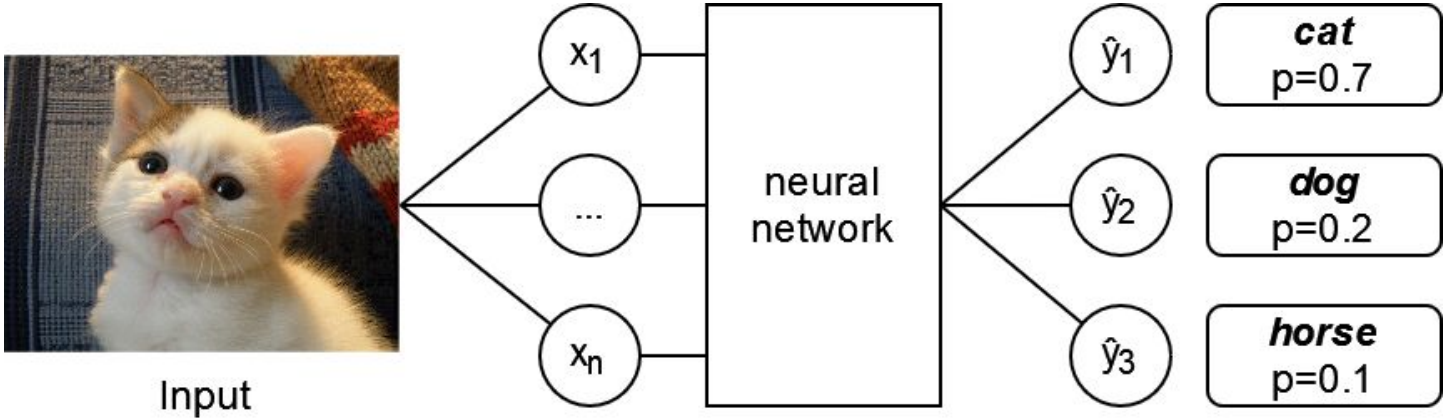
Building a Feedforward Network for Classification in PyTorch



Learning goals

- Create a multilayer neural network for classification in PyTorch
- Gain familiarity with the nn.Module syntax for network creation
- Understand usage of the softmax activation function
- Develop intuitions on Categorical Cross Entropy
- Use the Adam variant of stochastic gradient descent

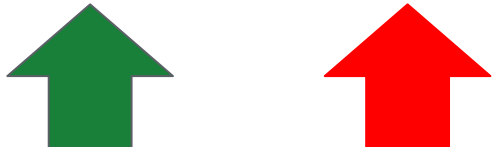
Multiclass classification: n probabilities for n classes



Classification $P(\hat{y}_{\text{cat}}) + P(\hat{y}_{\text{dog}}) + P(\hat{y}_{\text{horse}}) = 1$

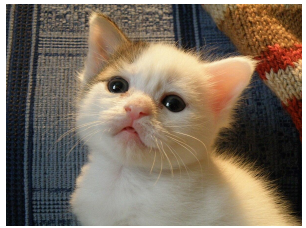
Cross entropy loss

$y = 1.0$ $\hat{y} = 0.75$



$$H(p,q) = -\frac{1}{n}\sum_{i=1}^{n} p_i(x)\log q_i(x)$$

Labels y map to $p_i(x)$, predictions $\hat{y}$ map to $q_i(x)$. Suppose that we show the network, only the following image (n=1).



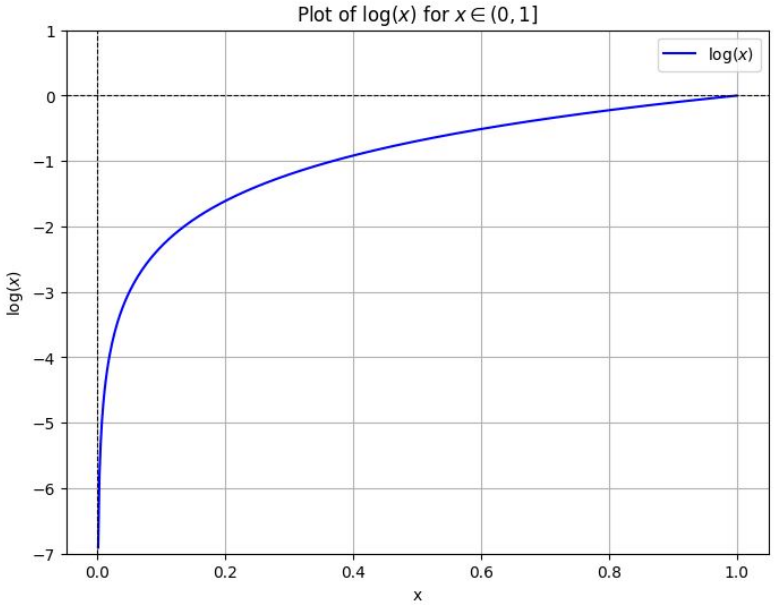
"one hot encoding" 🔥 = only one true label

The label for the image is encoded as y = [1, 0, 0] (i = 0 = cat, i = 1 = dog, i = 2 = horse)

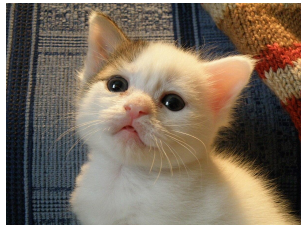
- First the network outputs **$\hat{y}$ = [0.75, 0.25, 0.0]**
- Then the network outputs **$\hat{y}$ = [0.99, 0.01, 0.0]** **(if trained correctly)**

Intuition for the log function in cross entropy loss

$$y = [1, 0, 0] \quad \text{where} \quad i = 0\ (\text{cat}),\ i = 1\ (\text{dog}),\ i = 2\ (\text{horse})$$



$$\ln(1) = \log(1) = 0$$



$$y_0 = 1 \quad \text{and} \quad \hat{y}_0 = 1$$

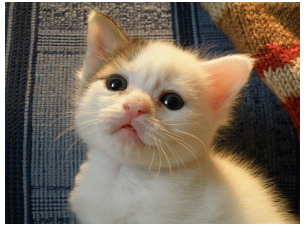
$$\text{Cross Entropy Loss} = -1 \cdot \log(1) = 0 \quad \text{(perfect prediction, no loss)}$$

$$y_0 = 1 \quad \text{and} \quad \hat{y}_0 = 0.01$$

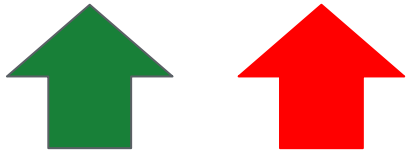
$$\text{Cross Entropy Loss} = -(1 \cdot \log(0.01)) = 4.65 \quad \text{(bad prediction, high loss)}$$

Implementing cross entropy loss from probabilities for a single sample

```python
import torch

# Example predicted probabilities from model
y_hat = torch.tensor([0.75, 0.25, 0.0])

# True label: y = [1, 0, 0]
y = torch.tensor([1, 0, 0], dtype=torch.float32)

def cross_entropy(y, y_hat):
    epsilon = 1e-15 # equal to 1 times 10^-15 (very small number)
    # Adding epsilon prevents log(0) which is undefined
    log_probs = torch.log(y_hat + epsilon)
    # Notice that the sum would only give a value different
    # than zero on the index of true_label
    # we are computing a single sample
    # so n = 1 (batch dimension)
    return -torch.sum(y * log_probs)

loss = cross_entropy(y, y_hat)
# Gives us tensor(0.287), not perfect, but not the worst possible
```



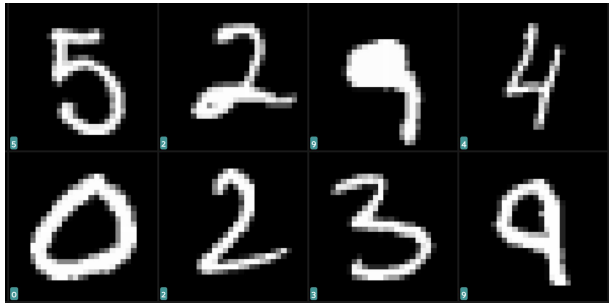
y = [1, 0, 0] y_hat = [0.75, 0.25, 0.0]



$$H(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} y_i \log \hat{y}_i$$

A minimal network for classification

```
import torch.nn as nn

# Model
model = nn.Sequential(

    # Flatten the input image
    nn.Flatten(),

    # The number of input features is the number of pixels in the image
    nn.Linear(in_features=28 * 28, out_features = 128),

    # We add a non-linearity
    nn.ReLU(),

    # We create 10 scores, aka 'logits', one for each class that we have
    # Notice that there is no ReLU after nn.Linear
    nn.Linear(in_features=128, out_features = 10)
)
```



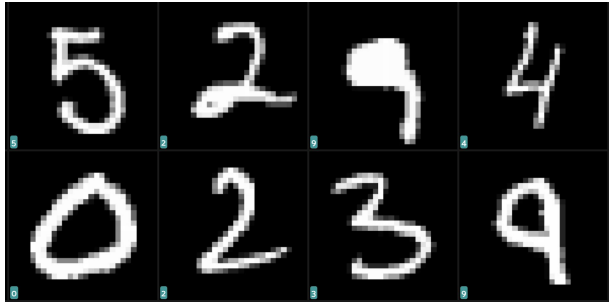
Tip: do not confuse these "logits" with the function described on https://en.wikipedia.org/wiki/Logit

$$f(x) = \ln\left(\frac{x}{1-x}\right)$$

❌ these "logits" are **not** this ^

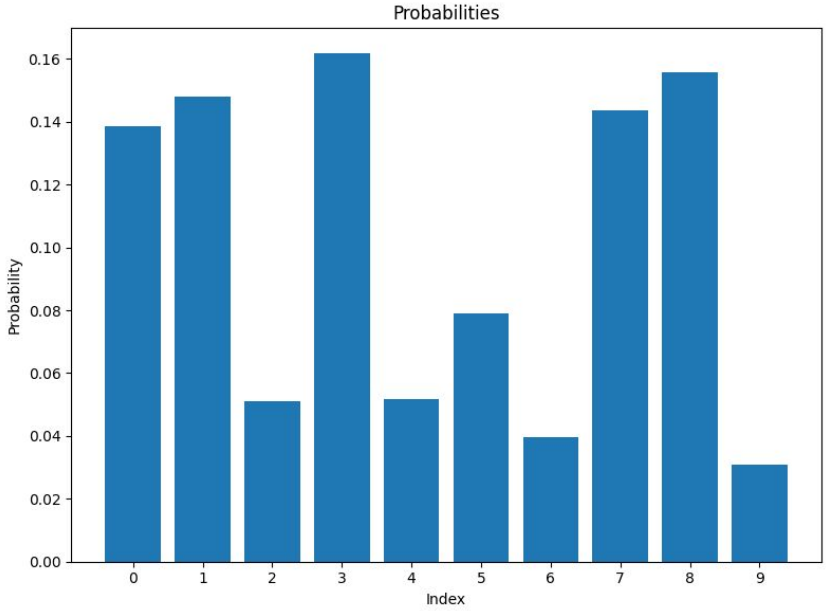
Using the nn.Module syntax

```
import torch.nn as nn

class MNISTClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        # Instead of Sequential, we define each module as a class attribute
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(in_features=28 * 28, out_features=128)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(in_features=128, out_features=10)

    def forward(self, x):
        # Define the forward pass and set a breakpoint
        import pdb; pdb.set_trace()
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x) # notice no relu here
        return x

# Create an instance of the model
model = MNISTClassifier()
```

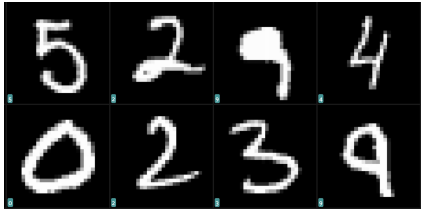


The softmax function

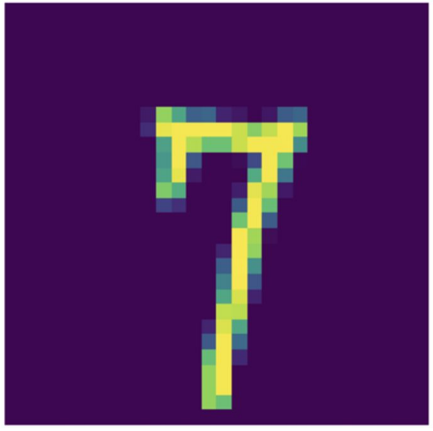
$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

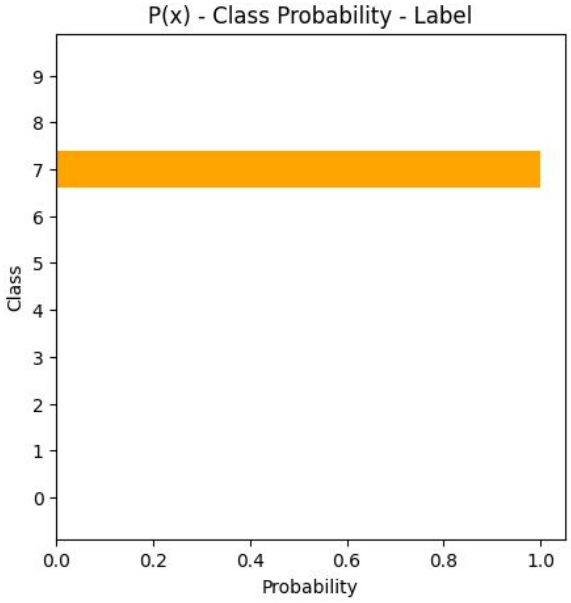


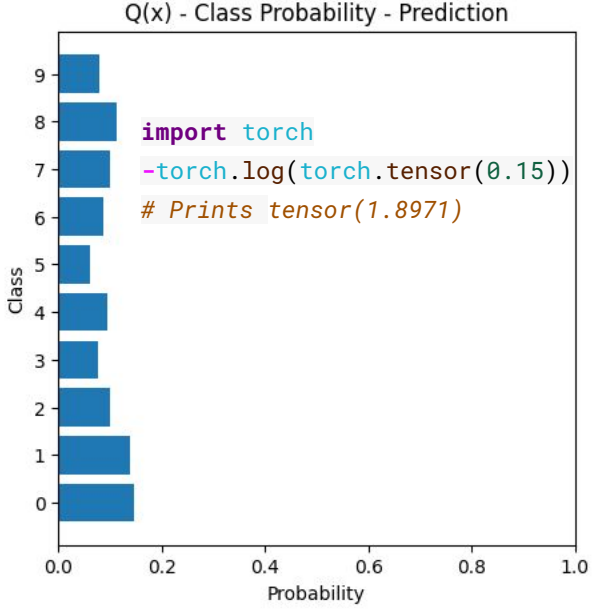

```python
import torch.nn.functional as F
logits = torch.tensor([ 0.7645,  0.8300, -0.2343,  0.9186, -0.2191,  0.2018, -0.4869,  0.8000, 0.8815, -0.7336])
probabilities = F.softmax(logits, dim=0)
print(probabilities.sum()) # prints tensor(1.00)
```



High cross entropy loss - high disagreement between P(x) = y and Q(x) = ŷ

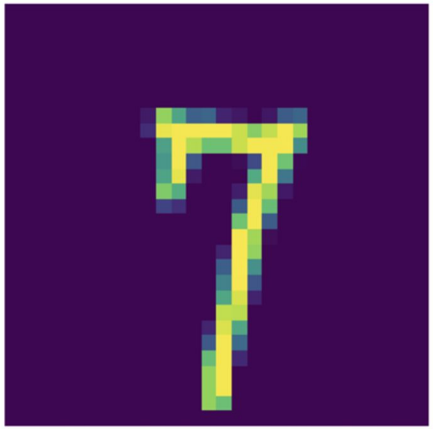


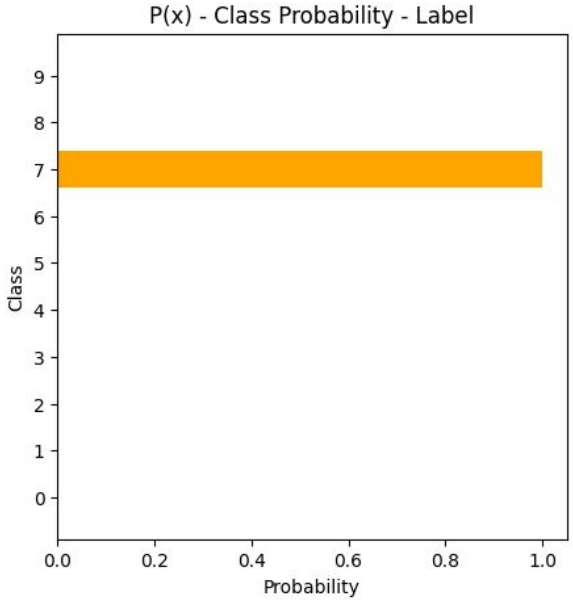


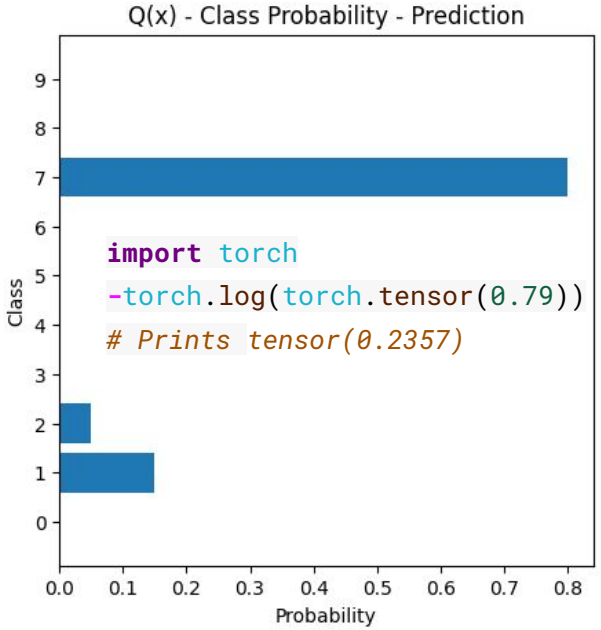


$$H(p,q) = -\frac{1}{n}\sum_{i=1}^{n} p_i(x)\log q_i(x) = H(y,\hat{y}) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log \hat{y}_i$$

Low cross entropy loss - low disagreement between y and ŷ







$$H(y, \hat{y}) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log \hat{y}_i$$

$$\text{Cross Entropy} = -\log(\hat{y}_i)$$

with index i being the one true class

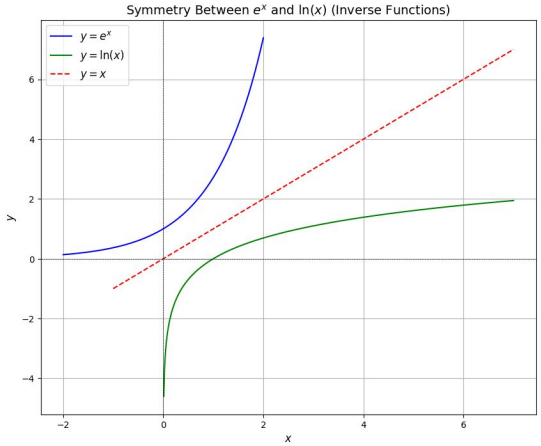
Quirks of nn.CrossEntropyLoss

```
# Compute output
logits = model(input_image)

# Turn output into probabilities, useful for interpretation
probabilities = F.softmax(logits)

# CrossEntropyLoss will compute log-softmax on the raw logits
# because it is more numerically stable
ce_loss = nn.CrossEntropyLoss()
ce_loss(logits, labels)
```

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \qquad \text{LogSoftmax}(\mathbf{x})_i = x_i - \log\left(\sum_{j=1}^{n} \exp(x_j)\right)$$

Turning LogSoftmax into interpretable probabilities

$$p_i = \exp(\log p_i) = e^{\ln p_i}$$




```
import torch.nn.functional as F
logits = torch.tensor([ 0.7645,  0.8300, -0.2343,
                        0.9186, -0.2191,  0.2018,
                        -0.4869,  0.8000, 0.8815,
                        -0.7336])
log_probabilities = F.log_softmax(logits, dim=0)
print(log_probabilities.sum()) # prints tensor (-24.6857)
probabilities = torch.exp(log_probabilities)
print(probabilities.sum()) # prints tensor(1.00)
```

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

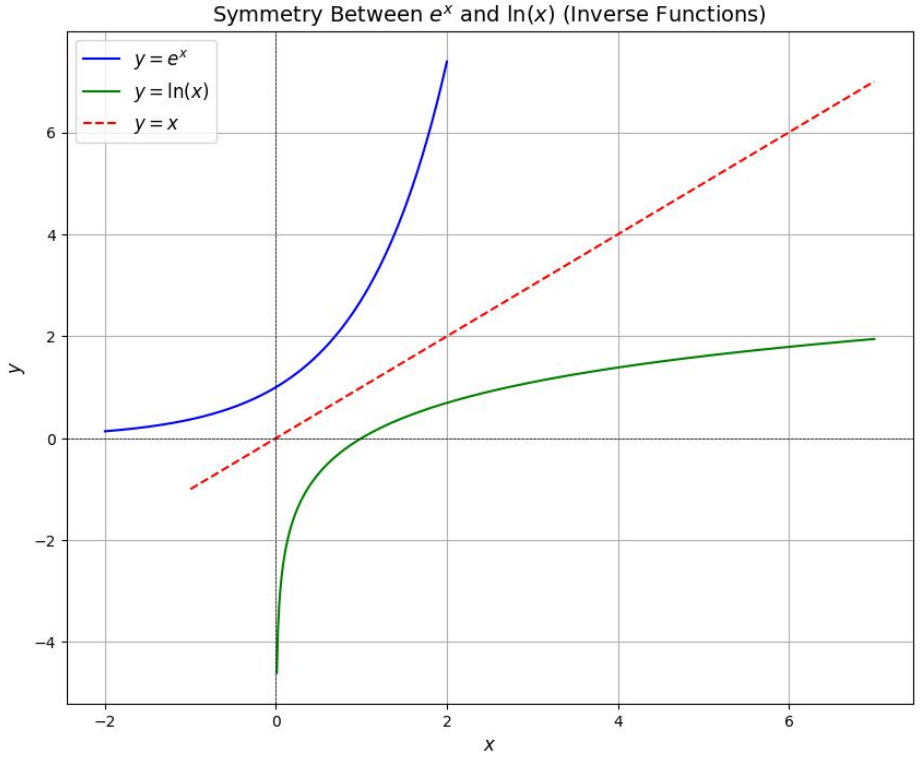
$$\text{LogSoftmax}(\mathbf{x})_i = x_i - \log\left(\sum_{} \exp(x_j)\right)$$

Why do we use log(x) = ln(x)?

$$p_i = \exp(\log p_i) = e^{\ln p_i}$$

$$\frac{d}{dx} e^x = e^x$$

$$\frac{d}{dx} \ln(x) = \frac{1}{x}$$



Cross entropy loss from logits

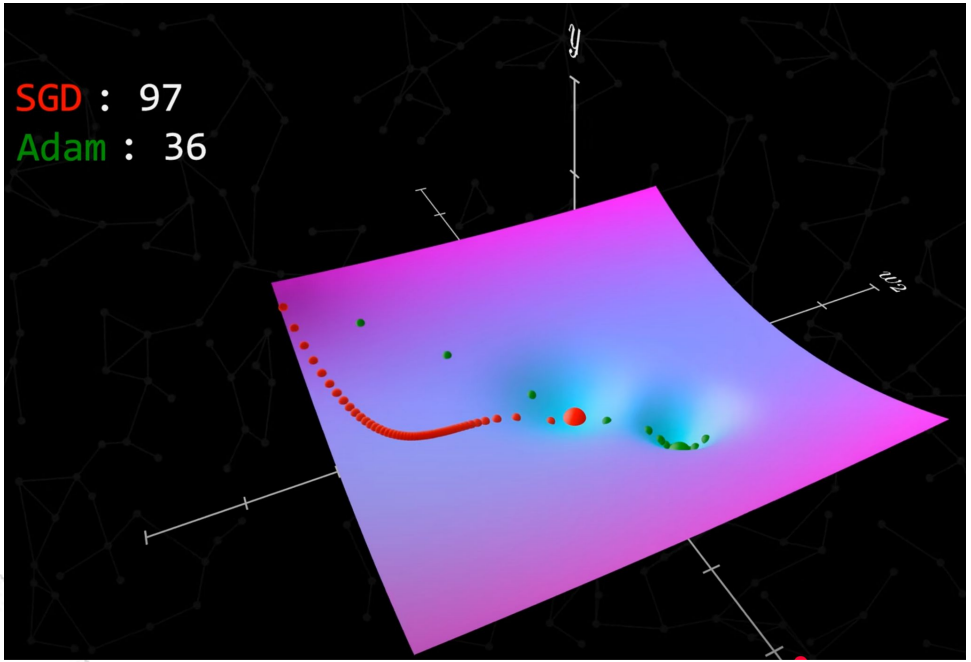
```python
import torch
import torch.nn.functional as F

def cross_entropy_loss(logits, labels):
    """
    Calculate cross entropy loss from logits using log softmax

    Args:
        logits: Tensor of shape (batch_size, num_classes) containing raw model outputs
        labels: Tensor of shape (batch_size,) containing class indices (0-9 for MNIST)

    Returns:
        loss: Scalar tensor with the mean loss
    """
    # Apply log softmax to get log probabilities
    log_probs = F.log_softmax(logits, dim=1)

    # Calculate negative log likelihood loss
    # This efficiently computes cross entropy without explicitly creating one-hot vectors
    loss = F.nll_loss(log_probs, labels)
```

“negative of the log of the likelihood (of the correct class” = F.nll_loss

$$\text{Cross Entropy} = -\log(\hat{y}_i)$$

with index i being the one true class

Extra: Adam vs Stochastic Gradient Descent



Adaptive Gradients with Momentum (ADAM) is variant of stochastic gradient descent that per well in many problems.

```
from torch.optim import Adam
optimizer = Adam(
            model.parameters(),
            lr=0.001)

# Compute gradients and update weights
ce_loss.backward()
optimizer.step()
```

Image from "Who is Adam and what is he optimizing?"

Summary

Feedforward neural networks for image classification

- Convert pixel inputs into class probabilities
- The number of output units determines the number of scores that we can produce
- The Adam optimizer is a variant of SGD that works well in practice

Classification pipeline

- We use one-hot encoding when we want to predict a single label per image
- We convert the raw output scores (logits) to probabilities using softmax

Cross entropy loss

- Measures classification prediction quality (agreement of probabilities in y hat and y)
- Used with log-softmax in PyTorch for numerical stability

Further reading

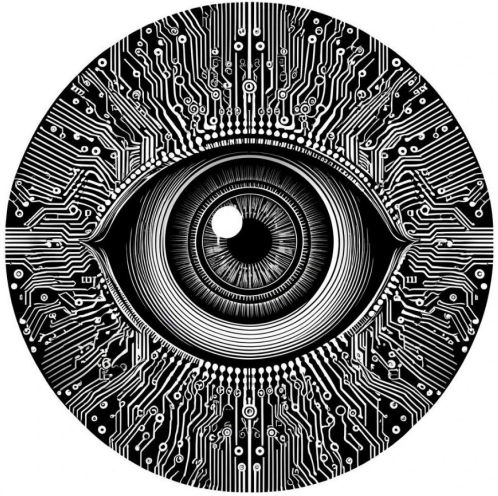
PyTorch's CrossEntropyLoss

- https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

Who is Adam and what is he optimizing?

- https://www.youtube.com/watch?v=MD2fYip6QsQ

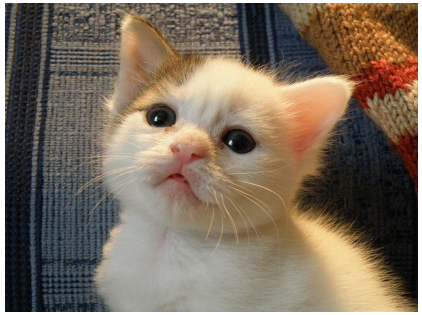
Performance Metrics for Classification and Experiment Tracking



Learning goals

- Understand the use of performance metrics for classification
- Identify elements in a confusion matrix
- Explore the effect of threshold moving on classification metrics
- Compute accuracy, precision, recall and f1 score for classifiers
- Checkpointing models with the best performance
- Logging experiments into `wandb`

Confusion matrices



$$y = P(\text{cat}) = 1.0$$

$$\hat{y} = P(\text{cat}) = 0.95$$

$$\begin{aligned}
\text{True Positive (TP)} &\Rightarrow y = 1, \hat{y} = 1 \\
\text{True Negative (TN)} &\Rightarrow y = 0, \hat{y} = 0 \\
\text{False Positive (FP)} &\Rightarrow y = 0, \hat{y} = 1 \\
\text{False Negative (FN)} &\Rightarrow y = 1, \hat{y} = 0
\end{aligned}$$

'positive' = 1, 'negative' = 0 (not interchangeable)

$$\begin{bmatrix} \text{TP: } 50 & \text{FN: } 10 \\ \text{FP: } 5 & \text{TN: } 100 \end{bmatrix}$$

A classifier outputs probabilities, so how do we define that y_hat = 1?

$$\text{Threshold for positive: } P > 0.5$$

Moving the threshold for ŷ = 1

Threshold for positive: $P > 0.5$

Threshold for positive: $P > 0.9$

Interpretability with performance metrics

Threshold for positive: $P > 0.5$

$$\text{accuracy} = \frac{\#\text{correct predictions}}{\#\text{predictions}}$$

assume that a classifier predicts the correct class with confidence 0.8 seven times and with confidence 0.1 three times

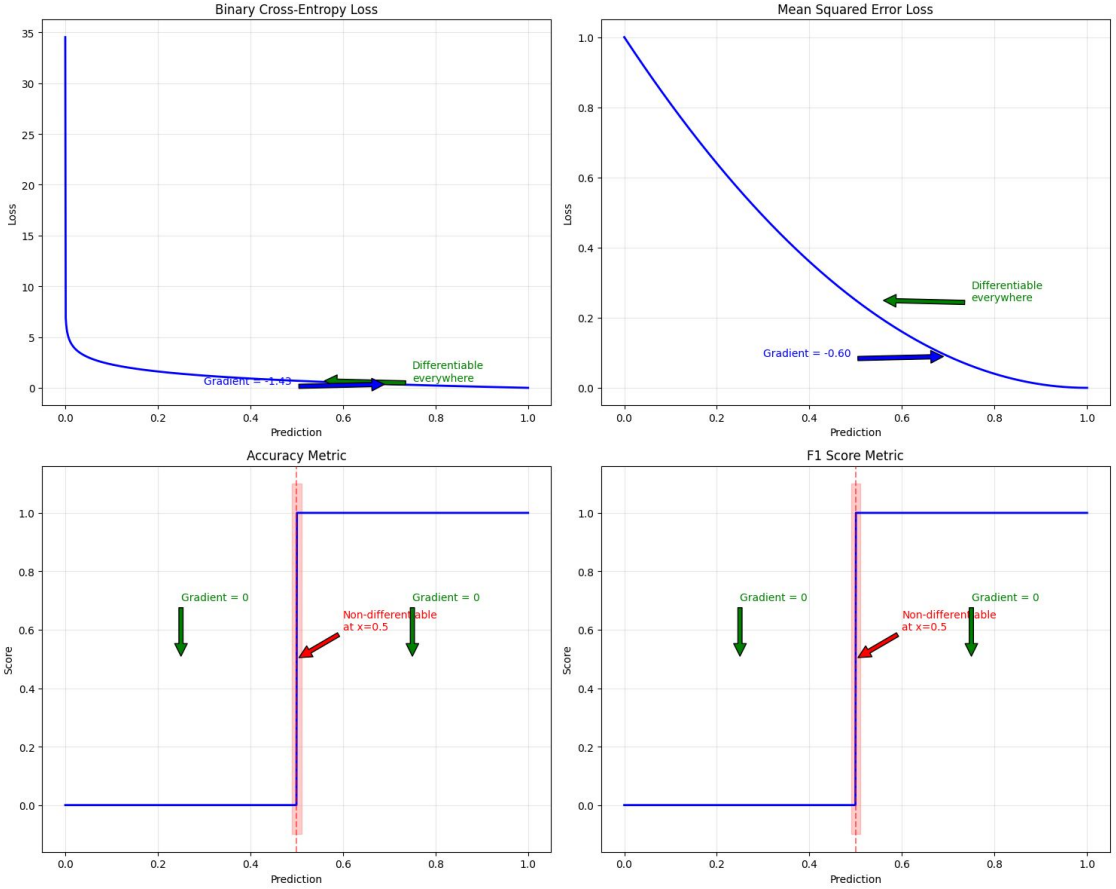
Cross Entropy Loss

$$-\frac{1}{10}\Big[7\ln(0.8) + 3\ln(0.1)\Big]$$

Numerically,

- $\ln(0.8) \approx -0.2231$, so $7\ln(0.8) \approx -1.5617$.
- $\ln(0.1) \approx -2.3026$, so $3\ln(0.1) \approx -6.9078$.
- Total is -8.4695; divided by 10, it's about 0.847.

Performance metrics vs loss functions

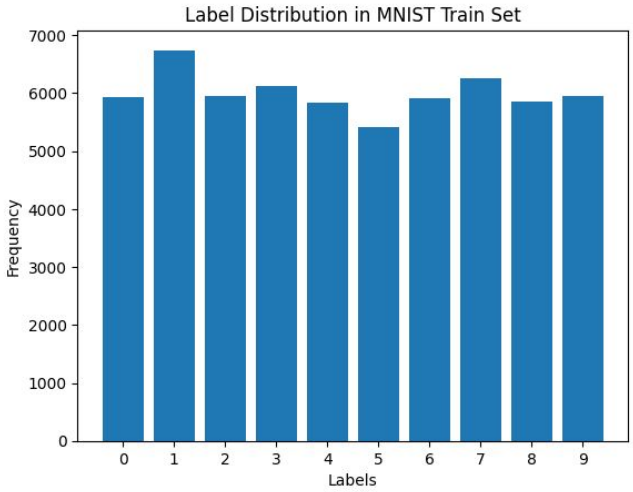


Gradient Examples:

```
BCE gradient at pred=0.7:
-1.429
MSE gradient at pred=0.7:
 -0.600
Accuracy 'gradient' at pred=0.7:
 0.000
F1 'gradient' at pred=0.7:
 0.000
```

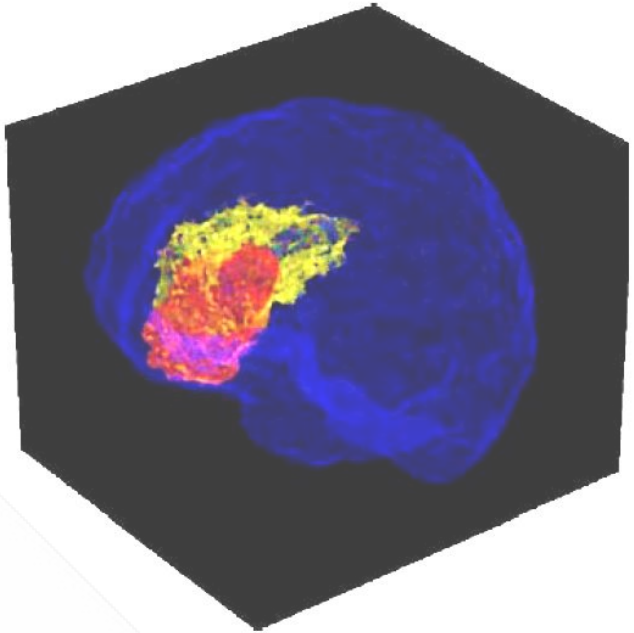
Multiclass accuracy is only good for balanced datasets

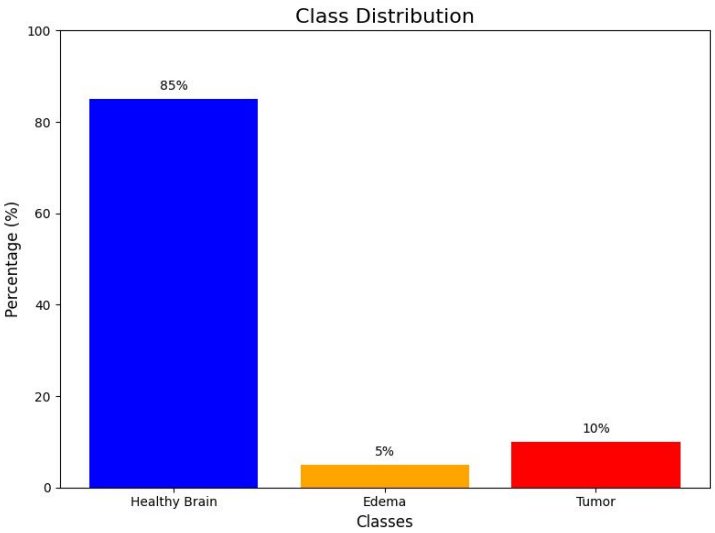
```python
def get_batch_accuracy(output, y):
    # Here we give as prediction
    # the label where we have the top confidence
    # No threshold, just top confidence
    pred = probs.argmax(dim=1, keepdim=True)
    correct = pred.eq(y.view_as(pred)).sum().item()
    return correct / probs.shape[0]
```



$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} = \frac{TP + TN}{TP + FP + FN + TN}$$

Accuracy is misleading in imbalanced datasets





'no tumor' in the image, majority class prediction

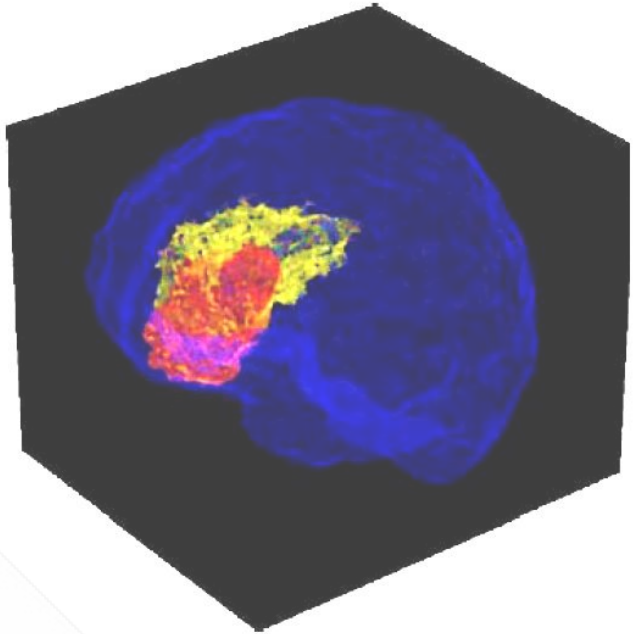
$$\text{Correct Predictions} = 900$$

$$\text{Total Predictions} = 1000$$

$$\text{Accuracy} = \frac{900}{1000} = 0.9\,(90\%)$$

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} = \frac{TP + TN}{TP + FP + FN + TN}$$

Tradeoffs in precision and recall



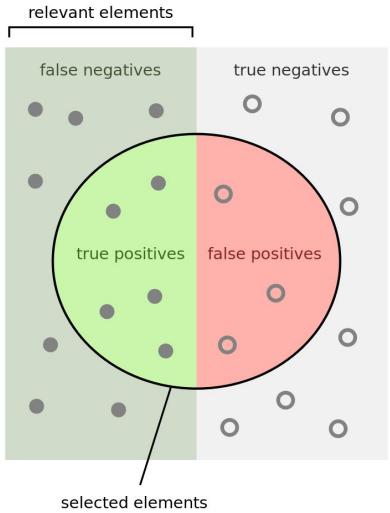
Labeling just one voxel as tumor, with the prediction being correct = **perfect precision**

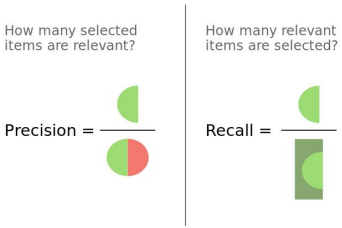
$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

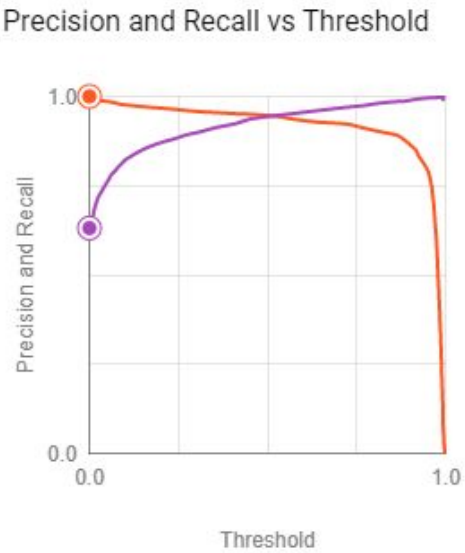
Labeling the whole brain as tumor = **perfect recall**

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Thresholds for y_hat = 1, precision, recall, and f1 metrics



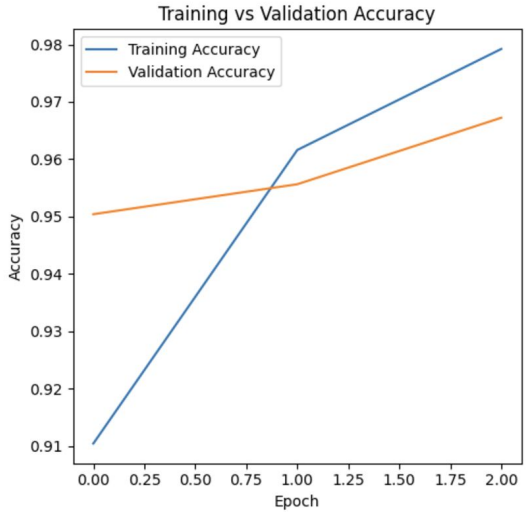




$$F_1 = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

Checkpointing models with the best performance

```
for epoch in range(epochs):

    train_loss, train_acc = train()
    valid_loss, valid_acc = validate()

    # Save model if validation
    #  accuracy improves
    if valid_acc > best_valid_accuracy:
        best_valid_accuracy = valid_acc

        # We save the model as a dictionary
        torch.save(model.state_dict(),
                   'best_model.pth')
```



Logging experiments and models on `wandb`

Image from wandb.ai

Summary

Confusion matrices are the basis of performance metrics for classification

- Threshold moving for the positive class affects them and makes the model more precise or sensitive
- The balance of a dataset impacts the metric used: accuracy for balanced datasets, precision and recall for imbalanced ones

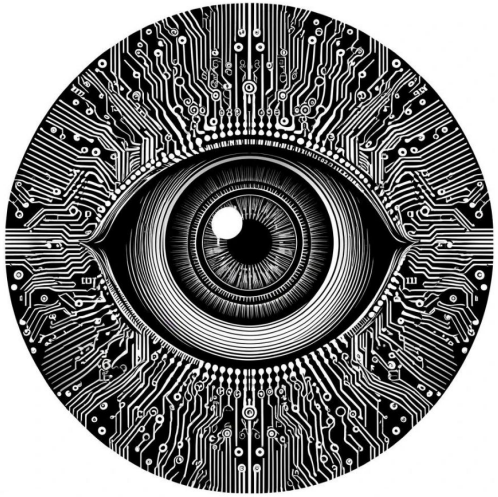
Performance metrics for classification are non-differentiable

- We use them for interpretability, not to directly update the weights

After aligning on a performance metric, we use it to checkpoint our models

- Experiment tracking allows us to benchmark our training runs

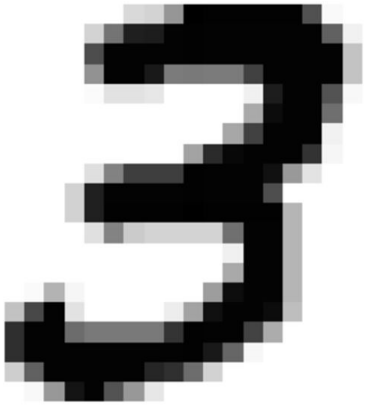
PyTorch Datasets and Dataloaders



Learning goals

- Create `Dataset` objects in PyTorch to wrap images and target values together
- Implement `DataLoader` PyTorch objects to feed data to a model
- Understand the connection between `DataLoader` and Stochastic Gradient Descent

The batch dimension is important for our models



`torch` image tensors follow the format: `[N, C, H, W]` where:

- `N` = batch size (number of images)
- `C` = channels (e.g., 1 for grayscale, 3 for RGB)
- `H` = height in pixels
- `W` = width in pixels

```
print(torch_tensor_gray.shape)

# torch.Size([1, 1, 28, 28])
```

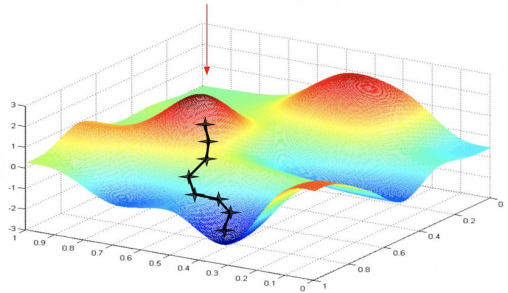
Feeding `torch` tensors to a PyTorch model

```
# Define the train loader with batch size and shuffling
batch_size = 32
train_loader = DataLoader(dataset=train_dataset,
                          batch_size=batch_size,
                          shuffle=True)

# Move model to device (GPU/CPU)
model = model.to(device)

# Training loop
for images, labels in train_loader:
    # Remember that data has to be explicitly sent to the GPU
    images, labels = images.to(device), labels.to(device)
    output = model(images)
    batch_loss = loss_function(output, labels)
```

Tip: use the nvidia-smi bash command to check the RAM available in your GPU

Shuffling the training set only

```
# Create data loaders
batch_size = 32
# Notice that we shuffle the training loader, but not the validation or test loaders.
# This practice of shuffling the training set is one of the techniques
# that have been shown to improve training.
train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(valid_subset, batch_size=batch_size, shuffle=False)
test_loader  = DataLoader(test_set,     batch_size=batch_size, shuffle=False)
```



$$w_{ij} = w_{ij} - (\text{learning rate} * \frac{dL}{dw_{ij}})$$

Dataloader depends on Dataset

```
# Define the MNIST Dataset class as a subclass of Dataset
class MNISTDataset(Dataset):
    # Content of the dataset
    # gets called on dataset_instance = MNISTDataset(<params>)
    def __init__(self, path_to_csv, labels=True, transform=None):
        …
    def __len__(self):
        # What gets called on len(dataset_instance)
        return len(self.data)

    def __getitem__(self, idx):
        # What gets called on dataset_instance[index]
        …
        return image_tensor, label_tensor
```

Practice creating a `Dataset`

Submit Prediction

KAGGLE · GETTING STARTED PREDICTION COMPETITION · ONGOING

Digit Recognizer

Learn computer vision fundamentals with the famous MNIST data

https://www.kaggle.com/competitions/digit-recognizer

References

Datasets and Dataloaders

- https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

MNIST on PyTorch's datasets

- https://pytorch.org/vision/0.20/generated/torchvision.datasets.MNIST.html

MNIST on Fiftyone

- https://try.fiftyone.ai/datasets/mnist/samples

Summary

Dataset objects wrap images and labels together

- Provide a standard way to access data through `__getitem__` and `__len__` methods

Dataloaders split datasets in batches

- Dataloaders shuffle the training data and maintain sequential order for validation and test sets

Dataloaders are fundamental to implement Stochastic Gradient Descent

- Their random sampling and shuffling of training samples provide the ‘stochastic’ part of Stochastic Gradient Descent

Resources

- Github Repository
- YouTube playlist
- Discord channel
  #practical-computer-vision-workshops