Ex09_v1.0

July 2, 2023

1 Probabilistic Machine Learning

1.1 Exercise Sheet No. 9 — DL Classifiaction on Binary MNIST

In the lecture, you have seen how to train a neural network. And in past tutorials, you trained a Gaussian Process on binary MNIST. In this tutorial, we will combine this knowledge to train a neural network on binary MNIST, and inspect some of the results.
See the Tasks and Evaluation Rules section for more details.

2 Imports and Helpers

[ ]:
```
# standard imports
import urllib.request  # to download MNIST
import gzip            # to download MNIST
from time import time

# Numerics
import jax
import jax.numpy as jnp
from jax.example_libraries import optimizers as jopt
import numpy as np
jax.config.update("jax_enable_x64", True)  # use double-precision numbers
jax.config.update("jax_platform_name", "cpu")  # we don't need GPU here

# Plotting
from matplotlib import pyplot as plt
from tueplots import bundles

plt.rcParams.update({"figure.dpi": 200})
plt.rcParams.update(bundles.beamer_moml())

import warnings
import logging

logging.getLogger('matplotlib').setLevel(level=logging.CRITICAL)
```

```
warnings.filterwarnings( "ignore", module = "matplotlib\..*" )


%config InlineBackend.figure_formats = ["png"]
%matplotlib inline
```

[ ]:
```
def inspect_batch(x_data, y_data, width=1.8, cmap="cividis", title=None):
    """
    Plot all given MNIST images with their corresponding labels.
    :param x_data: Numpy array of images with shape ``(b, h, w)``.
    :param y_data: Numpy array of labels with shape ``(b,)``
    :returns: Figure and axes.
    """
    num_axes = len(x_data)
    assert len(y_data) == num_axes, "Inconsistent inputs!"
    plt.rcParams.update(bundles.beamer_moml(rel_width=width))
    fig, axes = plt.subplots(ncols=num_axes)
    for i, ax in enumerate(axes):
        ax.imshow(x_data[i], cmap=cmap)
        ax.set_title(str(y_data[i]))
        ax.set_xticks([])
        ax.set_yticks([])
    if title is not None:
        fig.suptitle(title)
    return fig, axes
```

3 Training and Test Data

Since we aim to do binary classification and explore the model confidence, we will focus on two rather similar MNIST handwritten digits: 1 and 7. The following cell contains a convenience class that will allows us to download MNIST, store it persistently, and extract a binarized and standardized version.

[ ]:
```
class MNIST:
    """
    Static class to download MNIST into numpy arrays and extract a two-digit
    subset.
    """
    BASE_URL = "http://yann.lecun.com/exdb/mnist/"
    X_TRAIN_URL = "train-images-idx3-ubyte.gz"
    Y_TRAIN_URL = "train-labels-idx1-ubyte.gz"
    X_TEST_URL = "t10k-images-idx3-ubyte.gz"
    Y_TEST_URL = "t10k-labels-idx1-ubyte.gz"
    X_SHAPE = (28, 28)

    @classmethod
    def download(cls):
```

```
        """
        The MNIST dataset used in this notebook has been downloaded with this
        function. Returns a dict with the following ``np.uint8`` arrays:
        * x_train: (60000, 28, 28), y_train: (60000,)
        * x_test:  (10000, 28, 28), y_test:  (10000,)
        """
        x_train = urllib.request.urlopen(cls.BASE_URL + cls.X_TRAIN_URL).read()
        x_train = gzip.decompress(x_train)
        x_train = np.frombuffer(x_train, np.uint8, offset=16).reshape(
            -1, *cls.X_SHAPE)
        #
        y_train = urllib.request.urlopen(cls.BASE_URL + cls.Y_TRAIN_URL).read()
        y_train = gzip.decompress(y_train)
        y_train = np.frombuffer(y_train, np.uint8, offset=8)
        #
        x_test = urllib.request.urlopen(cls.BASE_URL + cls.X_TEST_URL).read()
        x_test = gzip.decompress(x_test)
        x_test = np.frombuffer(x_test, np.uint8, offset=16).reshape(
            -1, *cls.X_SHAPE)
        #
        y_test = urllib.request.urlopen(cls.BASE_URL + cls.Y_TEST_URL).read()
        y_test = gzip.decompress(y_test)
        y_test = np.frombuffer(y_test, np.uint8, offset=8)
        #
        return {"x_train": x_train, "y_train": y_train,
                "x_test": x_test, "y_test": y_test}

    @classmethod
    def extract_bmnist(cls, mnist, pos_digit=1, neg_digit=7,
                       standardize_imgs=True, dtype=np.float64):
        """
        :param mnist: The output of ``download``
        :param standardize_imgs: If true, returned images will have zero mean
          and unit variance.
        :param dtype: Ideally a large-resolution float.
        :returns: A dictionary that is a subset of the given ``mnist``, but
          only with ``pos_digit`` labeled as 1, and ``neg_digit`` labeled as 0.
        """
        # gather only desired digits, and label them +1, -1
        train_mask = (mnist["y_train"] == pos_digit) | (mnist["y_train"] ==
                                                         neg_digit)
        test_mask = (mnist["y_test"] == pos_digit) | (mnist["y_test"] ==
                                                       neg_digit)
        bmnist = {
            "x_train": mnist["x_train"][train_mask].astype(dtype),
            "y_train": ((mnist["y_train"][train_mask] == pos_digit)).
↪astype(dtype),
```

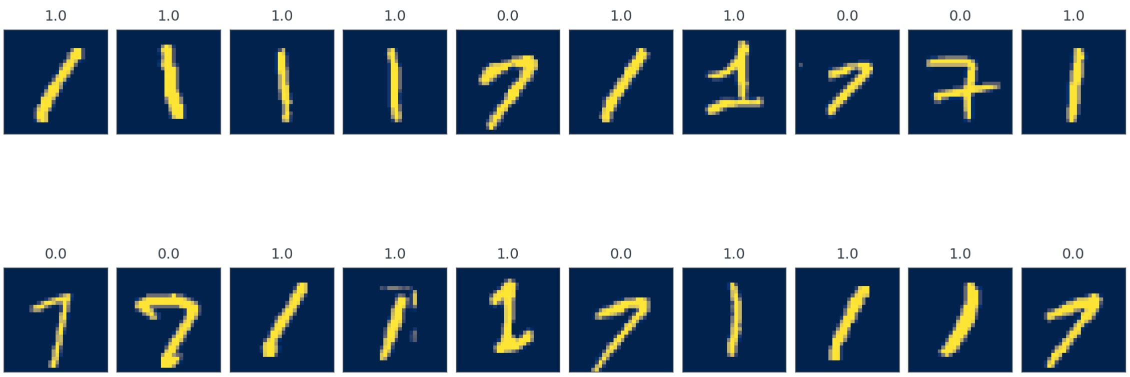
```
            "x_test": mnist["x_test"][test_mask].astype(dtype),
            "y_test": (mnist["y_test"][test_mask] == pos_digit).astype(dtype)}
        # sanity check
        len_x_train, len_y_train = len(bmnist["x_train"]),␣
↪len(bmnist["y_train"])
        len_x_test, len_y_test = len(bmnist["x_test"]), len(bmnist["y_test"])
        assert len_x_train == len_y_train, "Inconsistent training data in mnist?
↪"
        assert len_x_test == len_y_test, "Inconsistent test data in mnist?"
        # optionally standardize images
        if standardize_imgs:
            bmnist["x_train"] -= bmnist["x_train"].reshape(len_x_train, -1).
↪mean(axis=1)[:, None, None]
            bmnist["x_train"] /= bmnist["x_train"].reshape(len_x_train, -1).
↪std(axis=1)[:, None, None]
            bmnist["x_test"] -= bmnist["x_test"].reshape(len_x_test, -1).
↪mean(axis=1)[:, None, None]
            bmnist["x_test"] /= bmnist["x_test"].reshape(len_x_test, -1).
↪std(axis=1)[:, None, None]
        #
        return bmnist


# Attempt to recover preexisting mnist. If not preexisting, download anew and␣
↪save
%store -r mnist
try:
    mnist
    print("Fetched MNIST from storage!")
except NameError:
    print("Downloading MNIST...")
    mnist = MNIST.download()
    %store mnist
```

```
Fetched MNIST from storage!
```

[ ]:
```
POS_DIGIT, NEG_DIGIT = 1, 7  # feel free to play around with these, but stick␣
↪to (1, 7) for the submission
DTYPE = np.float64
bmnist = MNIST.extract_bmnist(mnist, POS_DIGIT, NEG_DIGIT, True, DTYPE)

inspect_samples = list(range(0, 10))
inspect_batch(bmnist["x_train"][inspect_samples],
              bmnist["y_train"][inspect_samples])

inspect_samples = list(range(10, 20))
inspect_batch(bmnist["x_train"][inspect_samples],
```

```
              bmnist["y_train"][inspect_samples]);
print("x_test.shape:", bmnist["x_test"].shape)
print("x_train.shape:", bmnist["x_train"].shape)
```

```
x_test.shape: (2163, 28, 28)
x_train.shape: (13007, 28, 28)
```



4 A Typical Deep Learning Setup:

As seen in the lecture slides, a typical DL training setup features the following components:

- **Dataloader:** Given is a dataset $\mathcal{D} = [(x_i, y_i)]_{i=1}^N$ that maps *inputs* x_i to *ground truth targets* y_i. We typically work with random subsets called *batches* $\mathcal{B} \overset{iid}{\sim} \mathcal{D}$. A dataloader has the function of providing said batches.
- **Model:** A *neural network* $\hat{y}_i = f(x_i, \theta)$ (with parameters θ), typically composed by many nonlinear, simple, parametrized, and differentiable functions called *layers*. It maps an input x_i, to a *predicted* output $\hat{y}_i$.
- **Initializer**: Setting the initial state for the model is also a relevant task. For simpler problems, like this one, it suffices to initialize the weights to small noise.
- **Objective**: The optimization *objective* in DL typically follows the *Empirical Risk Minimization* paradigm, featuring a *loss function* ℓ that penalizes differences between every $(y_i, \hat{y}_i)$ prediction-target pair, coupled with an additive *regularizer* ρ that does not depend on the data. In such cases, the objective $\mathcal{L}$ (also called *loss function*) has the form $\mathcal{L}(\theta) := \frac{1}{B} \sum_{i \in \{\mathcal{B}_1, \dots, \mathcal{B}_B\}} \{\ell(y_i, f(x_i, \theta))\} + \rho(\theta)$. Note how it only depends on the network parameters θ.
- **Optimizer**: The essence of DL training is to modify the weights θ in order to minimize $\mathcal{L}$, and to do so in a step-wise, batch-wise manner using gradient information (since f is differentiable, we can compute the derivatives of $\mathcal{L}$ with respect to θ, which tell us how to slightly modify θ in order to reduce $\mathcal{L}$). The optimizer is simply a component that has access to θ as well as such derivatives, and can update θ according to some heuristic (e.g. the *gradient descent* update is $\theta^{(t+1)} := \theta^{(t)} - \eta \nabla_{\theta^{(t)}} \mathcal{L}$ for some *learning rate* $\eta \in \mathbb{R}_{>0}$).

jax tip:

One major reason to use software libraries like jax for DL is that they compute the batch gradients automatically; we just need to define the "forward" computations using library components. Another advantage is that DL libraries also provide implementations for popular optimizers.

5 Tasks and Evaluation Rules:

In this tutorial, we will adapt the lecture example to MNIST, and analyze some of the obtained results. Specifically, the tasks are:

1. Define a training dataloader that provides `("x_train", "y_train")` batches, randomly drawn from $\mathcal{D}$ without replacement.
2. Define a two-class, ReLU, Multi-Layer Perceptron (like the one from the lecture) that maps MNIST images into a scalar, with dimensionalities `(784, 256, 64, 1)`.
3. Define the objective: Empirical Risk Minimization via cross-entropy loss coupled with *weight decay* (a.k.a. *L2 regularization*).
4. Complete the training and evaluation loop.
5. Once successfully trained, gather and plot the following data samples from the test set:

- The 5 "positive" examples with largest model output (i.e. clear positives)
- The 5 "negative" examples with smallest model output (i.e. clear negatives)
- The 5 "positive" examples with smallest model output (i.e. confusing positives)
- The 5 "negative" examples with largest model output (i.e. confusing negatives)

TUTORIAL EVALUATION RULES:

These tasks can be fulfilled with the already imported libraries, and no further libraries should be needed.

The cells below provide some scaffolding code that can be optionally used as a starting point (in which case the docstrings can be used as guidance, and the missing bits are signaled via NotImplemented, NotImplementedError and "TODO".

The Expected Result cells can be used as a guidance and to showcase correct functionality. In principle, they don't need to be modified, but it is allowed.

Code can be borrowed from the lectures, previous tutorials and other sources but it must be documented via docstrings and/or comments to show sufficient understanding of its interface and functionality (no blind copypaste allowed).

The trained model should surpass an accuracy of 95% after a few seconds on modest hardware.

The following hyperparameters allow to achieve that goal on modest hardware (provided as guidance, feel free to modify them):

[ ]:
```
# HYPERPARAMETERS

# model architecture and initialization
LAYER_SIZES = (784, 256, 64, 1)
INIT_STDDEV = 0.1
CLASSIFICATION_THRESHOLD = 0.5

# optimizer/objective
LEARNING_RATE = 1e-3
```

```
WEIGHT_DECAY = 1e-12

# training protocol
NUM_BATCHES = 5000 # This is an incorrect hard coding from the assumption that␣
↪we are using the full dataset. We are only using 1, 7 digits.
BATCH_SIZE = 25
RANDOM_SEED = 12345
```

[ ]:
```
# As the number of batches will depend on the size of training set and batch␣
↪size, we re-calculate it here.
NUM_TRAIN = bmnist['x_train'].shape[0]
NUM_COMPLETE_BATCHES, leftover = divmod(NUM_TRAIN, BATCH_SIZE)
NUM_BATCHES = NUM_COMPLETE_BATCHES + bool(leftover)
print("NUM_TRAIN: %d" % NUM_TRAIN)
print("NUM_COMPLETE_BATCHES: %d" % NUM_COMPLETE_BATCHES)
print("NUM_BATCHES: %d" % NUM_BATCHES)
```

```
NUM_TRAIN: 13007
NUM_COMPLETE_BATCHES: 520
NUM_BATCHES: 521
```

[ ]:
```
# We need to dynamically compute the number of batches based on the batch size
NUMBER_OF_BATCHES = NUM_BATCHES
```

6 Dataloaders

We provide a definition for the validation dataloader, which runs exactly once over the test subset. In contrast, `train_dataloader` should provide as many batches as desired, re-running over the dataset as many times as needed. For each run over the dataset, retrieved samples should be randomly sampled without replacement, and this randomness should be fully controlled via the random key `rng`: Running twice with same `rng` key should lead to same "random" batches, and different `rng` keys should lead to different "random" batches.

[ ]:
```
def train_dataloader(bmnist, batch_size=50, rng=jax.random.PRNGKey(12345),␣
↪debug=False):
    """
    Given a binary MNIST dataset, this generator runs infinitely, returning
    randomized batches from the training split.

    :param bmnist: Dictionary as returned by ``MNIST.extract_bmnist``
    :param rng: If a ``jax```random key is given, use it to shuffle
      all entries.
    :yields: An input-output pair of numpy arrays ``(x, y)``, where
      the first dimension of the arrays equals ``batch_size``,
      except for the last batch that may be smaller.
    """
    assert batch_size > 0, "batch_size <= 0 not supported"
```

```python
    x_train = bmnist["x_train"]
    y_train = bmnist["y_train"]
    training_set_size = len(x_train)
    # As we must compute the training set without replacement we created
    # a shuffled list of indices and extract ``batch_size`` elements
    # at a time from it.
    shuffled_indices = jax.random.permutation(rng, training_set_size)

    for batch_start_idx in range(0, training_set_size, batch_size):
        batch_indices = shuffled_indices[batch_start_idx : (batch_start_idx +␣
↪batch_size)]
        x = x_train[batch_indices]
        y = y_train[batch_indices]
        yield (x, y)


def test_dataloader(bmnist, batch_size=50):
    """
    Given a binary MNIST dataset, this generator runs once over its
    test split, in batched manner.

    :param bmnist: Dictionary as returned by ``MNIST.extract_bmnist``
    :yields: An input-output pair of numpy arrays ``(x, y)``, where
      the first dimension of the arrays equals ``batch_size``,
      except for the last batch that may be smaller.
    """
    assert batch_size > 0, "batch_size <= 0 not supported"
    for i in range(0, len(bmnist["x_test"]), batch_size):
        x = bmnist["x_test"][i : (i + batch_size), ...]
        y = bmnist["y_test"][i : (i + batch_size), ...]
        yield (x, y)
```

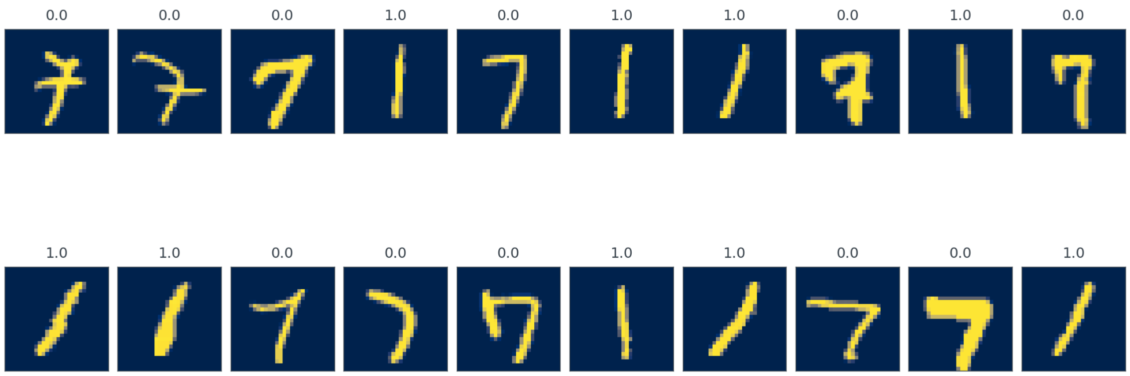
Let’s now inspect our dataloaders:

Expected result:

Running the cell below should plot 2 rows of 10 different random digits from the training set. Each digit should be correctly labeled (positive samples with a 1, negative samples with a 0), and all samples should be different. Using different seeds should lead to different results, whereas repeating seed should lead to same results.

[ ]:
```python
# Create a shuffled training dataloader with RANDOM_SEED and inspect
train_dl = train_dataloader(bmnist, BATCH_SIZE, rng=jax.random.
↪PRNGKey(RANDOM_SEED))
x_batch, y_batch = next(iter(train_dl))
inspect_samples = np.arange(10)
inspect_batch(x_batch[inspect_samples], y_batch[inspect_samples])
```

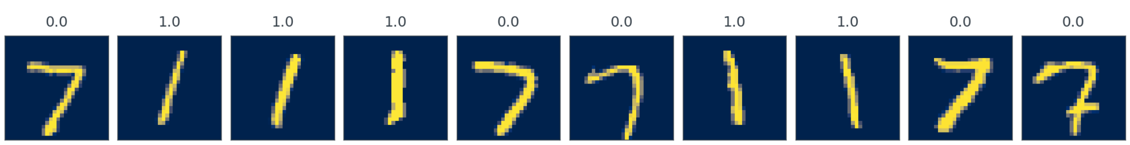
```
# Create a shuffled training dataloader with different seed and inspect
train_dl = train_dataloader(bmnist, BATCH_SIZE, rng=jax.random.
 ↪PRNGKey(RANDOM_SEED + 1))
x_batch, y_batch = next(iter(train_dl))
inspect_samples = np.arange(10)
inspect_batch(x_batch[inspect_samples], y_batch[inspect_samples]);
```



Expected result:

Running the cell below should plot a row of 10 digits from the test set. Each digit must also be correctly labeled.

[ ]:
```
# Create a test dataloader and inspect
test_dl = test_dataloader(bmnist, BATCH_SIZE)
x_batch, y_batch = next(iter(test_dl))
inspect_samples = np.arange(10)
inspect_batch(x_batch[inspect_samples], y_batch[inspect_samples]);
```



7 Model and Initialization

Define a two-class Multi-Layer Perceptron (like the one from the lecture) that maps MNIST images into a scalar, with dimensionalities `(784, 256, 64, 1)`. This means that it featuers 3 layers: one mapping from 784 dimensions to 256, and so on. Each i^{th} layer contains 2 parameters: a weight and a bias $[w_i, b_i]$, such that $x_{out} = \sigma(w^T x_{in}) + bias)$, where σ is a nonlinearity (in our case ReLU).

It should be using `jax`, in order to leverage automatic differentiation and batching.

jax tip:

Unlike other popular DL frameworks, jax follows a strictly functional paradigm, most notably meaning that the main building blocks are functions without state or side effects. Such functions expect all the input data to be passed through the (also stateless) function parameters, and all the results to be retrieved through the function results. A pure function will always return the same result if invoked with the same inputs. Not following this paradigm (e.g. by passing stateful computations to jax functions) is generally undefined and can lead to undesired behaviour. For us, this means that we will be writing functions that typically accept multiple inputs and outputs, and the "state" (e.g. current parameter values) will be stored in variables outside of those functions.

[ ]:
```
def mlp(params, inputs, nonlinearity=jax.nn.relu):
    """
    Computes the forward pass of an MLP, defined using JAX components. Note that
    it returns the *logits*. To map logits into predicted scores, a sigmoid
    function can be applied.

    :param params: List of pairs in the form ``[(w1, b1), (w2, b2), ...]`` where
      ``w_i, b_i`` are the weights and biases for layer ``i``, such that a layer
      computes ``outputs = nonlinearity((w_i @ inputs) + b_i)``.
    :param inputs: Batch of flattened input images with shape ``(batch,␣
↪in_shape)``
    :returns: A vector of shape ``(batch,)``, containing one logit per input␣
↪that
      should predict the corresponding binary class.
    """
    activations = inputs
    for w, b in params[:-1]:
        outputs = jnp.dot(activations, w) + b
        activations = nonlinearity(outputs)
    final_w, final_b = params[-1]
    logits = jnp.dot(activations, final_w) + final_b
    return logits[..., 0]


def create_mlp_params(layer_sizes, stddev=0.1, rng=jax.random.PRNGKey(12345)):
    """
    Creates MLP parameters of given sizes and initializes them with Gaussian
    noise of zero mean and given standard deviation.
    :param layer_sizes: List of integers in the form ``[d1, d2, ...]``,
      where each MLP layer maps from ``d_i`` dimensions to ``d_{i+1}``.
    :param stddev: Standard deviation of the initial Gaussian noise.
    :param rng: ``jax.random.PRNGKey`` to draw noise from.
    """
    params = []
    for m, n in zip(layer_sizes[:-1], layer_sizes[1:]):
        rng, rng_b = jax.random.split(rng)
        w = jax.random.normal(rng, (m, n)) * stddev
        b = jax.random.normal(rng_b, (n,)) * stddev
```

```python
        params.append((w, b))
    return params


def test_predictions(params, bmnist, batch_size=50, threshold=0.5):
    """
    Helper function to run ``sigmoid(model)`` over the whole test subset
    and compute the accuracy.

    :param threshold: Any sigmoid outputs above this number will be consider
      positive (i.e. a value of 1), otherwise negative (i.e. a value of 0).
    :param bmnist: See ``test_dataloader``.
    :param batch_size: See ``test_dataloader``.
    :returns: The triple ``(accuracy, logits, targets)``, where
      ``accuracy`` is the ratio of correctly classified samples, ``logits``
      are the predicted logits following the order provided by
      ``test_dataloader``, and ``targets`` are the corresponding ground
      truth annotations.
    """
    all_logits = []
    targets = []
    for x_batch, y_batch in test_dataloader(bmnist, batch_size):
        logits = mlp(params, x_batch.reshape(len(x_batch), -1))
        all_logits.extend(list(logits))
        targets.extend(list(y_batch))
    #
    predictions = jax.nn.sigmoid(np.array(all_logits)) > threshold
    targets = np.array(targets)
    accuracy = (predictions == targets).sum() / len(predictions)
    return accuracy, all_logits, targets
```

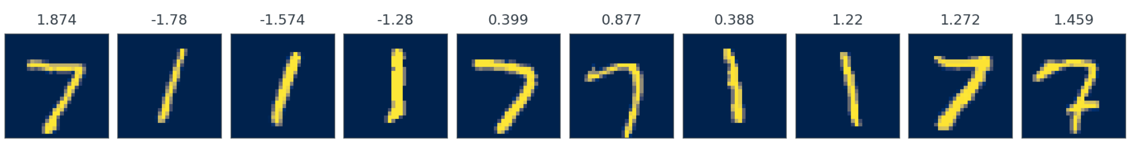
Let's inspect the model outputs when forward propagating through some samples.

Expected result:

Running the cell below should plot the same row of 10 digits from the test set as before (since the test dataloader is not random), but this time each digit is labeled with a noisy logit returned by the initialized (but not yet trained) MLP. Furthermore, a histogram of the accuracy over the whole test set should be plotted for 100 different random initializations, and the resulting distribution should be bell-shaped and centered around 50%.

[ ]:
```python
# create test dataloader
test_dl = test_dataloader(bmnist, BATCH_SIZE)
x_batch, y_batch = next(iter(test_dl))
# instantiate model
mlp_params = create_mlp_params(LAYER_SIZES, INIT_STDDEV,
                               jax.random.PRNGKey(RANDOM_SEED))
# forward pass
```

```python
predictions = mlp(mlp_params, x_batch.reshape(BATCH_SIZE, -1))
# plot 10 samples with their random logits
inspect_samples = np.arange(10)
inspect_batch(x_batch[inspect_samples],
              predictions[inspect_samples].round(decimals=3))

# compute test accuracy for 100 different random initializations
accuracies = []
for i in range(100):
    mlpp = create_mlp_params(LAYER_SIZES, INIT_STDDEV, jax.random.PRNGKey(RANDOM_SEED + i))
    acc, _, _ = test_predictions(mlpp, bmnist, BATCH_SIZE, CLASSIFICATION_THRESHOLD)
    accuracies.append(acc)
# plot histogram of accuracies
fig, ax = plt.subplots()
ax.hist(accuracies, bins=20)
ax.set_ylabel("Number of occurrences")
ax.set_xlabel("Test accuracy under random initialization");
```



8 Objective and Evaluation Metrics

In order to train the model using gradients, we must define the objective. Recall the formulation: $\mathcal{L}(\theta) := \frac{1}{B}\sum_{i\in\{\mathcal{B}_1,\dots,\mathcal{B}_B\}}\left\{\ell(y_i, f(x_i,\theta))\right\} + \rho(\theta)$

Here, the objective is the same as in the lecture: ℓ is the binary cross-entropy, and ρ is the L2 regularizer on all MLP parameters. Remember that `jax` follows a functional paradigm, where all relevant inputs and outputs must be stated explicitly and all side effects are kept outside of the functions.

[ ]:
```python
def regularizer(params, l2_reg=0.0):
    """
    :param params: Network parameters. See ``mlp`` docstring.
    :param l2_reg: Strength of the L2 regularization term.
    :returns: The L2 regularization term for the given parameters,
              ``(0.5 * l2_reg * l2norm(params)**2)``.
    """
    reg = 0.0
    for w, b in params:
        reg = reg + jnp.sum(w**2) + jnp.sum(b**2)
    return 0.5 * l2_reg * reg

def loss_fn(params, inputs, targets, l2_reg=0.0, debug=False):
    """
    :param params: Network parameters. See ``mlp`` docstring.
    :param inputs: Batch of network inputs. See ``mlp`` docstring.
    :param targets: Batch of ground truth annotations corresponding to␣
 ↪``inputs``,
      as provided by the dataloader.
    :param l2_reg: Strength of the L2 regularization term, such that
      ``result = cross_entropy + (0.5 * l2_reg * l2norm(params)**2)``.
    :returns: A single scalar representing the empirical risk plus the L2
      regularizer over the given batch, with respect to the given parameters.
    """
    predictions = mlp(params, inputs.reshape(inputs.shape[0], -1))

    assert predictions.shape == targets.shape
    assert inputs.shape[0] == targets.shape[0]

    cross_entropy_loss = jnp.logaddexp(0, -predictions * (2 * targets - 1)).
 ↪mean()
    regularization_term = regularizer(params, l2_reg)

    return cross_entropy_loss + regularization_term
```

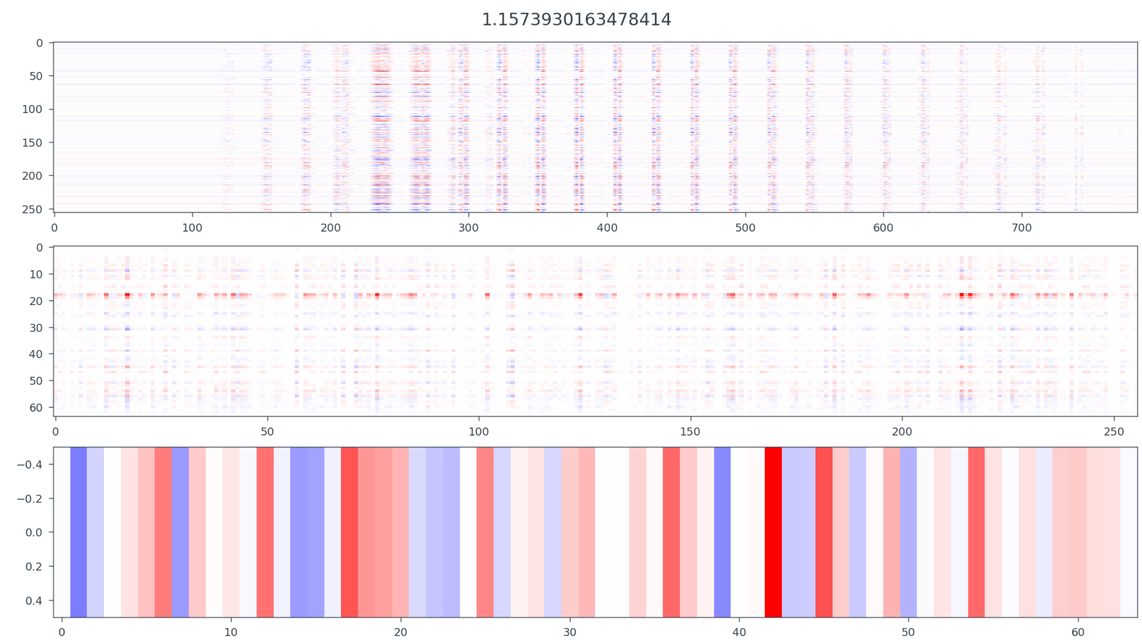
Let's re-run again the first test batch through the model, but this time we gather the loss and gradients.

Expected result:

Running the cell below should plot one image per MLP weight matrix, containing the corresponding gradients (which are also matrices of same shape). The loss value displayed at the top should be a positive float.

[ ]:
```python
# Create test dataloader
test_dl = test_dataloader(bmnist, BATCH_SIZE)
x_batch, y_batch = next(iter(test_dl))
```

```
# Instantiate model and forward+backward pass
mlp_params = create_mlp_params(LAYER_SIZES, INIT_STDDEV,
                               jax.random.PRNGKey(RANDOM_SEED))
loss, grads = jax.value_and_grad(loss_fn)(mlp_params, x_batch.
 ↪reshape(BATCH_SIZE, -1),
                                           y_batch, WEIGHT_DECAY)

# Plot weight gradients and loss
plt.rcParams.update(bundles.beamer_moml(rel_width=1.8, rel_height=1.8))
fig, axes = plt.subplots(nrows=len(grads))
fig.suptitle(loss)
for i, ax in enumerate(axes):
    wg = grads[i][0].T
    absmax = abs(wg).max()
    ax.imshow(wg, cmap="bwr", aspect="auto", vmin=-absmax, vmax=absmax)
```



9 Training Loop

We can finally put all pieces together to train the neural network. The only missing step is to actually update the parameters θ using the gradient information. This is the role of the optimizer (more info here). In point #3 of the cell below, we showcase how this is normally handled using `jax`. Note that we depart slightly from the functional paradigm:

jax tip:

For the optimization step, jax takes a notable exception from the functional paradigm via the so-called Just-In-Time (JIT) compilation. The idea is that, if we have a function that is expensive to run but has a "fixed" structure, we can speed up its computation substantially by allowing the JIT compiler to create an optimized version of it. The downside is that we lose flexibility: not everything can be JIT-compiled (e.g. `if-else` branching is generally not allowed), and some of the elements used get "frozen" during compilation, meaning the function becomes stateful (i.e. changing some Python variables after compilation won't alter the behaviour of already-compiled functions, which can cause some confusion).

For us, this means mostly two things:

1. The most expensive operations during DL training are typically the forward and backward comptuation, as well as the parameter update. We would like to bundle those into a single `update` function and and JIT-compile it.
2. But this function would basically depend on all other components. For this reason it needs to be defined right before the training loop starts. Also, it can not contain any dynamic structure like if-else branches.

Expected result:

Running the cell below should initialize and train our MLP for NUM_BATCHES and converge to less than 0.1 loss and over 90% accuracy after a few seconds. As training progresses, loss should generally decrease and accuracy increase.

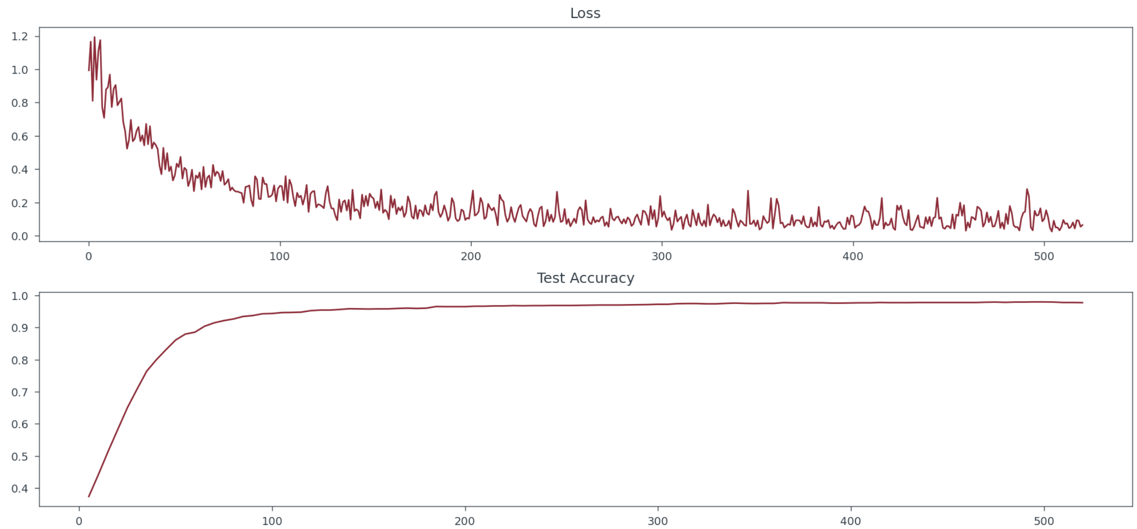
[ ]:
```
# 1. Dataloaders
train_dl = train_dataloader(bmnist, BATCH_SIZE, rng=jax.random.
 ↪PRNGKey(RANDOM_SEED))
test_dl = test_dataloader(bmnist, BATCH_SIZE)
# 2. Model params
mlp_params = create_mlp_params(LAYER_SIZES, INIT_STDDEV,
                               jax.random.PRNGKey(RANDOM_SEED))

# 3. Optimizer and JIT update step
opt_init, opt_update, get_params = jopt.sgd(LEARNING_RATE)
opt_state = opt_init(mlp_params)

@jax.jit
def update(step, opt_state, inputs, targets, l2_reg=0.0):
    """
    In order to speed up computations (not really necessary for small
    examples like this one, but crucial for larger DL setups), we
    "bundle" the forwardprop, backprop and update steps into a single
    JIT-able function.
    """
    value, grads = jax.value_and_grad(loss_fn)(get_params(opt_state),
                                               inputs, targets, l2_reg)
    opt_state = opt_update(step, grads, opt_state)
    return value, opt_state


losses, test_accs = [], []  # we will gather losses and accuracies
```

```
t0 = time()
# Training loop
for batch_t, (x_batch, y_batch) in enumerate(train_dl, 1):
    if batch_t > NUM_BATCHES:
        break
    loss, opt_state = update(batch_t, opt_state, x_batch, y_batch)

    losses.append(loss)
    # As the number of batches is approximately 500, we sample every 10% of the␣
↪batches.
    SAMPLE_EVERY = int(NUM_BATCHES*.01) # 200
    if batch_t % SAMPLE_EVERY == 0:
        test_acc, _, _ = test_predictions(get_params(opt_state), bmnist,␣
↪BATCH_SIZE, CLASSIFICATION_THRESHOLD)
        print(f"[step {batch_t:07d}] Loss={loss:5f}, Test accuracy={test_acc:
↪2f}")
        test_accs.append((batch_t, test_acc))
#
print("Elapsed seconds:", time() - t0)

plt.rcParams.update(bundles.beamer_moml(rel_width=1.8, rel_height=1.5))
fig, (ax_loss, ax_acc) = plt.subplots(nrows=2)
#
#ax_loss.plot(range(NUM_BATCHES), losses)
ax_loss.plot(range(len(losses)), losses)
ax_loss.set_title("Loss")
#
ax_acc.plot(*zip(*test_accs))
_ = ax_acc.set_title("Test Accuracy")
```

```
[step 0000005] Loss=0.937277, Test accuracy=0.373555
[step 0000010] Loss=0.878283, Test accuracy=0.442441
[step 0000015] Loss=0.905621, Test accuracy=0.514101
[step 0000020] Loss=0.628481, Test accuracy=0.582524
[step 0000025] Loss=0.581997, Test accuracy=0.650485
[step 0000030] Loss=0.543370, Test accuracy=0.708276
[step 0000035] Loss=0.560790, Test accuracy=0.764216
[step 0000040] Loss=0.528365, Test accuracy=0.799815
[step 0000045] Loss=0.332577, Test accuracy=0.831253
[step 0000050] Loss=0.343838, Test accuracy=0.861304
[step 0000055] Loss=0.397223, Test accuracy=0.879797
[step 0000060] Loss=0.278799, Test accuracy=0.885807
[step 0000065] Loss=0.289418, Test accuracy=0.904300
[step 0000070] Loss=0.328932, Test accuracy=0.914933
[step 0000075] Loss=0.272330, Test accuracy=0.921868
[step 0000080] Loss=0.260314, Test accuracy=0.926953
[step 0000085] Loss=0.302371, Test accuracy=0.934813
```

```
[step 0000090] Loss=0.222269, Test accuracy=0.937587
[step 0000095] Loss=0.232983, Test accuracy=0.943135
[step 0000100] Loss=0.283252, Test accuracy=0.944059
[step 0000105] Loss=0.198958, Test accuracy=0.946833
[step 0000110] Loss=0.258279, Test accuracy=0.947295
[step 0000115] Loss=0.305747, Test accuracy=0.948220
[step 0000120] Loss=0.171107, Test accuracy=0.952843
[step 0000125] Loss=0.254378, Test accuracy=0.954693
[step 0000130] Loss=0.121713, Test accuracy=0.954693
[step 0000135] Loss=0.212727, Test accuracy=0.956542
[step 0000140] Loss=0.148493, Test accuracy=0.958853
[step 0000145] Loss=0.176348, Test accuracy=0.958391
[step 0000150] Loss=0.224607, Test accuracy=0.957929
[step 0000155] Loss=0.138057, Test accuracy=0.958391
[step 0000160] Loss=0.169601, Test accuracy=0.958391
[step 0000165] Loss=0.176581, Test accuracy=0.959778
[step 0000170] Loss=0.116873, Test accuracy=0.960703
[step 0000175] Loss=0.150290, Test accuracy=0.959778
[step 0000180] Loss=0.191112, Test accuracy=0.960703
[step 0000185] Loss=0.112982, Test accuracy=0.965788
[step 0000190] Loss=0.115496, Test accuracy=0.965326
[step 0000195] Loss=0.097450, Test accuracy=0.965326
[step 0000200] Loss=0.103010, Test accuracy=0.965326
[step 0000205] Loss=0.145530, Test accuracy=0.966713
[step 0000210] Loss=0.191752, Test accuracy=0.966713
[step 0000215] Loss=0.063485, Test accuracy=0.967638
[step 0000220] Loss=0.084354, Test accuracy=0.967638
[step 0000225] Loss=0.128984, Test accuracy=0.968562
[step 0000230] Loss=0.085656, Test accuracy=0.968100
[step 0000235] Loss=0.056305, Test accuracy=0.968562
[step 0000240] Loss=0.085798, Test accuracy=0.968562
[step 0000245] Loss=0.113167, Test accuracy=0.969025
[step 0000250] Loss=0.160954, Test accuracy=0.969025
[step 0000255] Loss=0.103718, Test accuracy=0.969025
[step 0000260] Loss=0.065540, Test accuracy=0.969487
[step 0000265] Loss=0.112259, Test accuracy=0.969949
[step 0000270] Loss=0.115674, Test accuracy=0.970411
[step 0000275] Loss=0.104777, Test accuracy=0.970411
[step 0000280] Loss=0.074083, Test accuracy=0.970411
[step 0000285] Loss=0.059846, Test accuracy=0.970874
[step 0000290] Loss=0.066021, Test accuracy=0.971336
[step 0000295] Loss=0.179033, Test accuracy=0.971798
[step 0000300] Loss=0.239205, Test accuracy=0.972723
[step 0000305] Loss=0.109212, Test accuracy=0.972723
[step 0000310] Loss=0.101825, Test accuracy=0.974572
[step 0000315] Loss=0.068923, Test accuracy=0.975035
[step 0000320] Loss=0.135395, Test accuracy=0.975035
[step 0000325] Loss=0.187686, Test accuracy=0.974110
```

```
[step 0000330] Loss=0.080908, Test accuracy=0.974110
[step 0000335] Loss=0.064220, Test accuracy=0.975497
[step 0000340] Loss=0.090124, Test accuracy=0.976422
[step 0000345] Loss=0.060223, Test accuracy=0.975497
[step 0000350] Loss=0.058033, Test accuracy=0.975035
[step 0000355] Loss=0.094709, Test accuracy=0.975497
[step 0000360] Loss=0.128703, Test accuracy=0.975497
[step 0000365] Loss=0.049341, Test accuracy=0.977809
[step 0000370] Loss=0.059819, Test accuracy=0.977346
[step 0000375] Loss=0.117961, Test accuracy=0.977346
[step 0000380] Loss=0.056692, Test accuracy=0.977346
[step 0000385] Loss=0.054180, Test accuracy=0.977346
[step 0000390] Loss=0.061805, Test accuracy=0.976422
[step 0000395] Loss=0.057607, Test accuracy=0.976422
[step 0000400] Loss=0.122704, Test accuracy=0.976884
[step 0000405] Loss=0.081031, Test accuracy=0.977346
[step 0000410] Loss=0.108535, Test accuracy=0.977346
[step 0000415] Loss=0.108337, Test accuracy=0.978271
[step 0000420] Loss=0.068314, Test accuracy=0.977809
[step 0000425] Loss=0.155124, Test accuracy=0.977809
[step 0000430] Loss=0.154896, Test accuracy=0.977809
[step 0000435] Loss=0.123132, Test accuracy=0.978271
[step 0000440] Loss=0.071876, Test accuracy=0.978271
[step 0000445] Loss=0.228434, Test accuracy=0.978271
[step 0000450] Loss=0.059415, Test accuracy=0.978271
[step 0000455] Loss=0.127493, Test accuracy=0.978271
[step 0000460] Loss=0.031032, Test accuracy=0.978271
[step 0000465] Loss=0.095357, Test accuracy=0.978271
[step 0000470] Loss=0.076689, Test accuracy=0.979196
[step 0000475] Loss=0.058100, Test accuracy=0.979658
[step 0000480] Loss=0.041210, Test accuracy=0.978733
[step 0000485] Loss=0.062753, Test accuracy=0.979658
[step 0000490] Loss=0.137853, Test accuracy=0.979658
[step 0000495] Loss=0.033181, Test accuracy=0.980120
[step 0000500] Loss=0.086998, Test accuracy=0.980120
[step 0000505] Loss=0.025817, Test accuracy=0.979658
[step 0000510] Loss=0.052679, Test accuracy=0.978271
[step 0000515] Loss=0.053971, Test accuracy=0.978271
[step 0000520] Loss=0.055045, Test accuracy=0.977809
Elapsed seconds: 18.84768795967102
```



10 Inspect Trained Results

Once successfully trained, gather and plot the following data samples from the test set:

- The 5 "positive" examples with largest model output (i.e. clear positives)
- The 5 "negative" examples with smallest model output (i.e. clear negatives)
- The 5 "positive" examples with smallest model output (i.e. confusing positives)
- The 5 "negative" examples with largest model output (i.e. confusing negatives)

[ ]:
```
# Gather all required data
test_acc, logits, targets = test_predictions(
    get_params(opt_state), bmnist, BATCH_SIZE, CLASSIFICATION_THRESHOLD)
predictions = np.asarray(jax.nn.sigmoid(np.array(logits)))
assert test_acc > 0.9, "Low accuracy! Has the model been correctly trained?"


def retrieve_interesting_samples(predictions, targets, num_samples=5):
    """
    :param predictions: Numpy array of ``sigmoid(mlp(x_i))`` floats.
    :param targets: Numpy array of ground truth scalars ``y_i`` given in
      same order as predictions.
    :returns: A dictionary ``{"posmax": [idx1, idx2, ...], "posmin": [...],
      "negmax": [...], "negmin": [...]}`` with the indexes for the N
      labeled "positive" examples with largest prediction, the N "positive"
      examples with smallest model output, the N "negative" examples with
      largest model output and the N "negative" examples with smallest model
      output, where N is ``num_samples``.
    """
    def selected_indices_min_max(selected_indices):
```

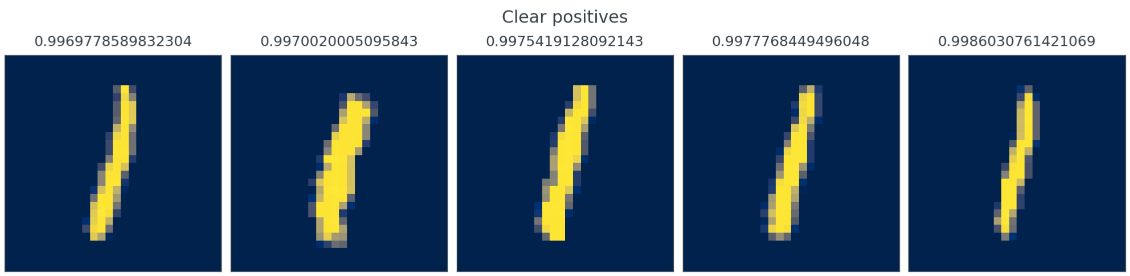
```
    """
    :param selected_indices: Numpy array of indexes of selected samples
    :returns: A dictionary ``{"min": [idx1, idx2, ...], "max": [...]}`` with␣
↪the ```num_samples```
              smallest and largest indexes.
    """
    selected_predictions = predictions[selected_indices]
    selected_to_original_idx = dict((selected_idx, original_idx) for␣
↪selected_idx, original_idx  in enumerate(selected_indices))
    selected_predictions_sorted = np.argsort(selected_predictions)
    top_idx = selected_predictions_sorted[-num_samples:]
    bottom_idx  = selected_predictions_sorted[:num_samples]
    top_original_idx = [selected_to_original_idx[idx] for idx in top_idx]
    bottom_original_idx = [selected_to_original_idx[idx] for idx in␣
↪bottom_idx]
    assert jnp.allclose(predictions[top_original_idx],␣
↪selected_predictions[top_idx])
    assert jnp.allclose(predictions[bottom_original_idx],␣
↪selected_predictions[bottom_idx])
    return {"min": bottom_original_idx, "max": top_original_idx}

  positive_indices = np.where(targets == 1)[0]
  negative_indices = np.where(targets == 0)[0]
  pos_min_max = selected_indices_min_max(positive_indices)
  neg_min_max = selected_indices_min_max(negative_indices)

  return {"posmax": pos_min_max["max"] ,
          "posmin": pos_min_max["min"] ,
          "negmax": neg_min_max["max"] ,
          "negmin": neg_min_max["min"] }
```

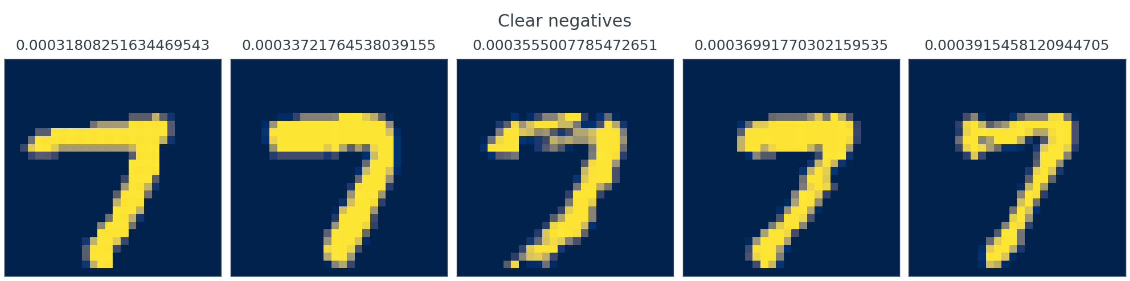
Expected result:

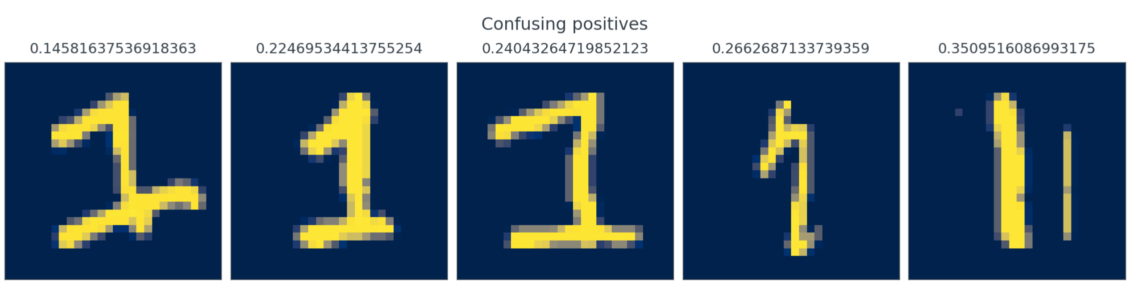
Running the cell below should gather indexes for the 4 interesting groups of test samples, as described above. Then, each group should be plotted in its own row, where each row contains all samples of the same class. The “clear” rows should depict instances that are clearly identifiable, whereas the “confusing” rows should depict examples that present some irregularities.

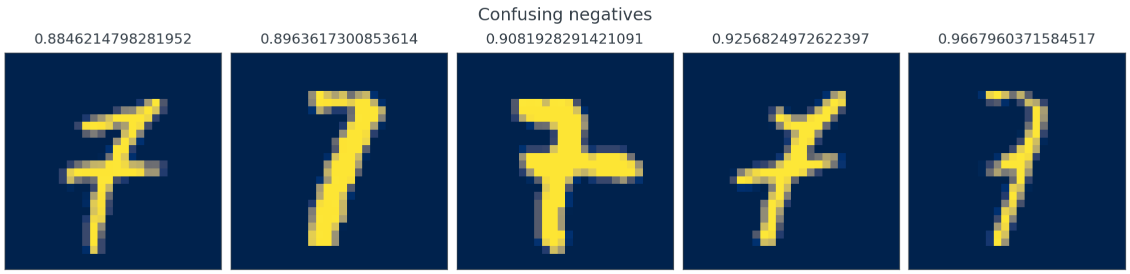
[ ]:
```
# gather interesting samples
interesting = retrieve_interesting_samples(predictions, targets, 5)

# Plot "clear" examples
inspect_batch(bmnist["x_test"][interesting["posmax"]],␣
↪predictions[interesting["posmax"]],
              title="Clear positives")

inspect_batch(bmnist["x_test"][interesting["negmin"]],␣
↪predictions[interesting["negmin"]],
```

```
              title="Clear negatives")

# Plot "confusing" examples
inspect_batch(bmnist["x_test"][interesting["posmin"]], predictions[interesting["posmin"]],
              title="Confusing positives")

inspect_batch(bmnist["x_test"][interesting["negmax"]], predictions[interesting["negmax"]],
              title="Confusing negatives");
```









End of `Exercise Sheet No. 9 - DL Classifiaction on Binary MNIST`.