1. Start with one perceptron

Work individually. Use NumPy for array operations and Matplotlib for plots. In Tasks 1-8, write the forward pass, losses, gradients, updates and optimizers yourself. Keep one notebook named STUDENTID_CV_Lab10.ipynb. Each task asks you to understand, implement, change and explain; no scheduled demonstrations are required.

Understand the concept

A perceptron is a small decision rule. It multiplies each input by a weight, adds a bias, and compares the resulting score with zero. A weight controls how one input affects the score. The bias moves the boundary without changing the input. The score z and the predicted class are different quantities.

Small worked example

For x = [1,1], w = [1,1] and b = -1.5, the score is z = 1 + 1 - 1.5 = 0.5. Because z >= 0, predict class 1. For x = [0,1], the score is -0.5, so predict 0. Complete the other two rows yourself.

Implement and experiment

1. Enter the four inputs and both label columns below. AND is 1 only when both inputs are 1. XOR is 1 only when the inputs differ.
2. Write perceptron(X,w,b), returning all scores and hard-threshold predictions. Print one row per input for AND.
3. Keep w fixed and change b to -0.5. Then restore b=-1.5 and change only w[0] to -1. Predict which rows change before running them.
4. Scatter the four inputs, colour them by label, and draw the score-zero boundary.

Coding hint - complete the missing work

```
import numpy as np
import matplotlib.pyplot as plt
X = np.array([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])
y_and = np.array([[0.],[0.],[0.],[1.]])
y_xor = np.array([[0.],[1.],[1.],[0.]])
w = np.array([[1.],[1.]])
b = -1.5
# TODO: z = ...          # use X @ w + b
# TODO: predicted = ...  # convert z >= 0 to integers
```

Check and debug

Scores and predictions must have shape (4,1). With the original w and b, your predictions must match AND. For a non-vertical boundary use x2 = -(w1*x1+b)/w2; if w2=0, draw a vertical line instead.

Explain what you learned

Why does a bias change move several decisions together, while a weight change depends on the corresponding input? Write one sentence about what a perceptron learns.

2. From a decision to a probability

Understand the concept

A hard step jumps directly from class 0 to class 1. Its derivative is zero away from the jump and undefined at the jump, so it is unsuitable for ordinary gradient training. A sigmoid is smooth: p = 1/(1+exp(-z)). It produces a number between 0 and 1. We train using p, then predict class 1 when p >= 0.5. Hidden activations such as tanh and ReLU create intermediate features; they are not necessarily class probabilities.

Small worked example

If z = 0, sigmoid(z) = 0.5. For a true label y = 1 and predicted probability p = 0.8, binary cross-entropy is -ln(0.8), about 0.2231. If that same example instead receives p = 0.2, its loss is about 1.6094. The confident wrong prediction costs more.

Implement and experiment

1. Implement sigmoid, tanh and ReLU using NumPy. Plot them and the hard step over z from -3 to 3 on the same labelled axes.
2. Write binary_loss(p,y) for mean binary cross-entropy. Use the formula in the hint and clip a separate probability copy to [1e-12,1-1e-12] before taking logs.
3. For y=1 and then y=0, calculate loss at p=0.1, 0.5 and 0.9. Print a six-row table. Also calculate mean absolute error and mean squared error on these probability predictions.
4. Pass the perceptron scores from Task 1 through sigmoid, then threshold at 0.5. Compare with the hard-step predictions.

Coding hint - complete the missing work

```
def sigmoid(z):
    # TODO: translate 1 / (1 + exp(-z))
    raise NotImplementedError

def binary_loss(p, y):
    safe_p = np.clip(p, 1e-12, 1 - 1e-12)
    # TODO: average -(y*log(safe_p)+(1-y)*log(1-safe_p))
    raise NotImplementedError

# tanh(z) = np.tanh(z); ReLU(z) = np.maximum(z, 0)
# Derivatives for later: sigmoid'=p*(1-p)
# tanh'=1-h*h; ReLU'=1 where z>0, otherwise 0
```

Check and debug

At p=0.5 the binary loss is about 0.6931 for either label. Loss is a scalar average over examples. Keep y as a column array to avoid accidental (N,N) broadcasting. Clipping protects logarithms; it is not a new class-decision rule.

Explain what you learned

Can two correct class predictions have different losses? Why can a hidden activation be negative even though a probability cannot? Explain what each quantity measures.

3. Write gradient descent by hand

Understand the concept

A gradient tells us how a small parameter increase changes the loss locally. Gradient descent subtracts a learning-rate-scaled gradient. Weights and bias are trainable parameters, so both must change. For a sigmoid neuron trained with mean binary cross-entropy, the chain rule simplifies the gradient to the equations below. N is the number of examples used for this update.

Small worked example

Use one example x=[1,1], y=1, w=[0,0], b=0. Then p=0.5 and p-y=-0.5. The weight gradient is [-0.5,-0.5] and the bias gradient is -0.5. With learning rate 0.1, the new weights are [0.05,0.05] and the new bias is 0.05. Compute the new probability and check that the loss decreases.

Implement and experiment

1. Implement neuron_gradients(X,y,w,b). Compute the full forward pass first, then dw and db using the same old parameter values.
2. Verify the one-example worked update. Next write a loop that trains on all four AND rows; record loss before training and after each update.
3. Run 500 updates at learning rates 0.01, 0.1 and 1. Start each run from w=0 and b=0. Plot the three loss histories and print the final predictions.
4. Run one deliberate wrong-sign update (adding the gradient), compare the loss change, then restore subtraction.

Coding hint - complete the missing work

```
# Mean-loss gradients:
# p = sigmoid(X @ w + b)
# dw = X.T @ (p-y) / len(X)
# db = np.mean(p-y)

def fit_neuron(X, y, eta, steps):
    w = np.zeros((X.shape[1], 1)); b = 0.0
    history = []
    for step in range(steps):
        # TODO: compute p, loss, dw and db
        # TODO: update w and b AFTER computing both gradients
        pass
    return w, b, history
```

Check and debug

dw has the same shape as w. Divide by the batch size exactly once; do not average dw a second time. Print the first update so you can trace it. A useful rate depends on the input scale and loss; a larger rate need not be better.

Explain what you learned

Why does subtracting a negative gradient increase a weight? What changed across the three learning rates, and why? What does a loss curve tell you that accuracy alone may hide?

4. Why add a hidden layer?

Understand the concept

One sigmoid neuron still has a straight boundary: p=0.5 exactly when its score is zero. XOR places the two class-1 points at opposite corners, so a single straight line cannot separate the classes. A hidden layer learns several intermediate features and combines them. A 2-4-1 network has two inputs, four hidden activations and one output probability. Use tanh in the hidden layer and sigmoid at the output.

Small worked example

Consider a smaller hidden response H=[0.5,-0.5], output weights W2=[[1],[-1]] and output bias 0. Its output score is 1, so the probability is about 0.7311. The hidden features are not class labels: the output layer must still combine them. Your full model will learn its own hidden features.

Implement and experiment

1. First reuse fit_neuron on XOR. Try two random initializations and plot the boundary with the labelled points. Explain why changing the learning rate cannot remove the straight-line limitation.
2. Initialize the 2-4-1 parameters with seed 7, normal weights of SD=0.5 and zero biases. Write forward_hidden(X,params); do not train it yet.
3. Print Z1, H, output scores and P for the four XOR points. Print every shape and calculate one hidden activation by hand.
4. Set one hidden unit's input weights to zero on a copy. Inspect which activations and output probabilities change.

Coding hint - complete the missing work

```
rng = np.random.default_rng(7)
params = {"W1": rng.normal(0, 0.5, (2,4)),
          "b1": np.zeros((1,4)),
          "W2": rng.normal(0, 0.5, (4,1)),
          "b2": np.zeros((1,1))}

def forward_hidden(X, params):
    # TODO: Z1 = X @ W1 + b1
    # TODO: H = tanh(Z1)
    # TODO: scores = H @ W2 + b2; P = sigmoid(scores)
    # Return P and a cache containing X, Z1 and H.
    raise NotImplementedError
```

Check and debug

For four rows: X=(4,2), W1=(2,4), H=(4,4), W2=(4,1), P=(4,1). The network contains 17 trainable numbers, including biases. A random untrained hidden layer is not guaranteed to classify XOR correctly.

Explain what you learned

How does the role of a hidden feature differ from an output probability? Which representational limitation of one neuron does the hidden layer address?

5. Backpropagate through both layers

Understand the concept

The output error must reach the weights that created the hidden features. Backpropagation applies the chain rule from the output toward the input. D2 is the derivative of the mean loss with respect to output scores. Multiplying by W2.T moves that signal backward; multiplying by the tanh derivative accounts for the hidden nonlinearity. Cache the forward values instead of changing them during differentiation.

Small worked example

For one example, H=[0.5,-0.5], W2=[[1],[-1]], output bias 0 and y=1 give P=0.7311. Then D2 is about -0.2689, dW2 is approximately [-0.1345,0.1345] as a column, and db2=-0.2689. The hidden-score gradient is about [-0.2017,0.2017]. Recalculate these values before using them as a check.

Implement and experiment

1. Implement backward_hidden using the equations below; return all four gradient arrays. Match every gradient shape to its parameter.
2. Compute all gradients, then update all parameters. Write a 5000-update training loop for XOR at learning rate 0.1, starting from Task 4's original initialization. Plot loss and print probabilities and predictions.
3. Check one W1 and one W2 entry numerically: [L(theta+eps)-L(theta-eps)]/(2*eps), with eps=1e-5. Restore the original scalar after each measurement.
4. Temporarily omit the tanh derivative in the backward pass. Observe whether your numerical check detects the bug, then restore it.

Coding hint - complete the missing work

```
# N = len(X); all values come from the SAME forward pass.
D2  = (P-y) / N
# TODO: dW2 = H.T @ D2; db2 = sum over rows
D1  = (D2 @ params["W2"].T) * (1-H**2)
# TODO: dW1 = X.T @ D1; db1 = sum over rows
# Bias sums: np.sum(D, axis=0, keepdims=True)
# Only AFTER all four gradients exist:
# for key in params: params[key] -= eta * gradients[key]
```

Check and debug

A numerical check compares the derivative of the SAME mean loss with one analytical gradient entry. Use relative error abs(a-n)/(1e-8+abs(a)+abs(n)); aim below 1e-4, and inspect absolute error near zero. Updating W2 before computing D1 mixes old and new parameter states.

Explain what you learned

Why does changing a first-layer weight affect the final loss? Why can code run without errors even when its gradients are wrong? Use your deliberate bug as evidence.

6. Generalize to several layers

Understand the concept

A deeper network repeats the same operation: a linear combination followed by an activation. Store a list of layers rather than writing a new formula for each depth. During the forward pass, save A[0]=X, each hidden activation, and each preactivation Z. During the backward pass, visit layers in reverse order. Without nonlinear activations, multiple affine layers collapse to one affine map.

Small worked example

The architecture 2-4-4-1 has parameter shapes (2,4), (4,4) and (4,1), plus bias rows of lengths 4, 4 and 1. Its parameter count is (8+4)+(16+4)+(4+1)=37. On the backward pass, each D must have one row per example and one column per unit in that layer.

Implement and experiment

1. Write init_layers(widths,seed=7,scale=0.5), forward_layers and backward_layers for a list of layers. Use tanh in hidden layers and sigmoid at the output.
2. Train 2-4-4-1 on XOR using your own gradient-descent loop. Check one gradient in every layer with finite differences before comparing results.
3. Compare hidden widths 2 and 8. Next compare all-zero with small random weight initialization. Reset parameters between trials.
4. Remove the hidden nonlinearities on a separate copy. Compare its XOR result with the nonlinear model. Keep the output sigmoid.

Coding hint - complete the missing work

```
# For layer l, with input activation A[l]:
# dW[l] = A[l].T @ D[l]
# db[l] = D[l].sum(axis=0, keepdims=True)
# For the PREVIOUS tanh hidden layer:
# D_prev = (D[l] @ W[l].T) * (1 - A[l]**2)

def backward_layers(X, y, layers, cache):
    # TODO: start from (P-y)/N at the output
    # TODO: iterate reversed(range(len(layers)))
    # TODO: cache all gradients; do not update inside this loop
    raise NotImplementedError
```

Check and debug

At each layer, print input, weight, activation and gradient shapes. Equal zero weights give identical hidden units; on balanced XOR, an all-zero tanh network can have zero useful gradients. Random initialization breaks this symmetry; it is not just cosmetic.

Explain what you learned

Why do earlier layers receive different gradients after adding a hidden layer? Did more width or depth help your run? Why is “more layers always means more expressive” false when every hidden operation is linear?

7. Use your manual network on images

Understand the concept

An image is a grid of measurements. Flattening makes each pixel an input feature, so the same layers and gradients can learn from images. The synthetic images below contain a bright horizontal line (label 0) or vertical line (label 1) at different positions. Independent training and validation sets let us check whether the learned rule works on new examples, rather than only the training images.

Small worked example

A 6 x 6 image becomes a vector of length 36. A 36-12-1 network has 36*12+12+12*1+1=457 trainable parameters. Changing the number of input features changes matrix sizes, not the gradient-descent principle.

Implement and experiment

1. Run the supplied generator and display eight labelled images. Use your own 36-12-1 network, with weight SD=0.25, tanh or ReLU hidden units and a sigmoid output.
2. Train with manual full-batch gradient descent. Start with learning rate 0.5 and 300 updates; record train and validation losses. If using ReLU, implement its derivative and check gradients away from zero.
3. Compare widths 4 and 32, then tanh versus ReLU, one factor at a time. Save independent copies of the checkpoint with lowest validation cross-entropy.

Coding hint - complete the missing work

```
def make_images(n, seed):
    r = np.random.default_rng(seed)
    images = r.uniform(0.0, 0.15, (n,6,6))
    labels = np.arange(n) % 2
    for i, label in enumerate(labels):
        pos = r.integers(0, 6)
        if label == 0: images[i, pos, :] += 0.8
        else: images[i, :, pos] += 0.8
    order = r.permutation(n)
    return np.clip(images[order],0,1), labels[order,None]

train_images, y_train = make_images(480, 10)
val_images, y_val = make_images(120, 20)
X_train = train_images.reshape(480,36)
X_val = val_images.reshape(120,36)
# Keep make_images(120,30) for the final test only.
```

Check and debug

Do not fit parameters on validation examples. Keep the same split and initialization rule across comparisons. Saving best_layers=layers shares objects; copy every weight and bias array. A best checkpoint at the final update is valid if validation loss keeps falling.

Explain what you learned

What changed when the input became pixels? Did better training loss also give better validation loss? Explain underfitting or overfitting only if your observed curves support it.

8. Implement the optimizers yourself

Understand the concept

Backpropagation calculates a gradient; an optimizer decides how to use it. Ordinary gradient descent uses only the current gradient. Momentum carries an accumulated direction from previous updates. Adam keeps moving averages of both signed gradients and squared gradients, corrects their initial zero bias, and scales each parameter's update. Each weight and bias needs its own state.

Small worked example

Use L(theta)=(theta-2)^2, starting theta=0. Its gradient is 2*(theta-2)=-4. With eta=0.1, the first GD and the specified momentum updates both give theta=0.4. Adam's corrected first averages are mhat=-4 and shat=16, so its first update gives approximately theta=0.1. Calculate the second update for each method yourself.

Implement and experiment

1. Write gd_step, momentum_step and adam_step for every parameter array. Use the formulas below. Initialize each state array to zeros with the parameter's shape.
2. Run five updates on the scalar example and print step, theta, gradient and loss. Trace the stored momentum/Adam state at the first two updates.
3. Use each optimizer with the SAME XOR architecture, batches, initialization and update budget. Compare learning rates 0.01 and 0.1 and plot loss curves. Reset optimizer memory for every fresh run.
4. Optional: reuse your backward pass on mini-batches of size 2, dividing by the current mini-batch size, not the full dataset size.

Coding hint - complete the missing work

```
# g is the gradient from the current backward pass.
# GD: theta = theta - eta*g
# Momentum: v = 0.9*v + g; theta = theta - eta*v
# Adam:
# m = 0.9*m + 0.1*g
# s = 0.999*s + 0.001*g*g
# mhat = m/(1-0.9**t); shat = s/(1-0.999**t)
# theta = theta - eta*mhat/(sqrt(shat)+1e-8)
# t starts at 1 and increases ONCE per batch update.
# TODO: implement these rules inside your parameter loop.
```

Check and debug

For Adam, t is shared across the parameters updated in one batch; do not advance it once per weight matrix. State arrays must not be shared accidentally across layers. A comparison at one learning rate does not prove one optimizer is always best.

Explain what you learned

What does each optimizer remember, and how does that change its second update? Why should the state be reset when starting a new model? Use your curves to explain your result.

9. Meet PyTorch and torch

What are these names?

PyTorch is a Python library for building and training neural networks. **torch** is the Python package name used to access PyTorch: writing **import torch** loads it. In this lab, they refer to the same library, not two separate tools.

A **tensor** is an array of numbers, similar to a NumPy array. PyTorch can track tensor calculations and calculate derivatives automatically; this feature is called **autograd**. Setting requires_grad=True asks it to track gradients. Calling loss.backward() calculates those gradients; it does not update the weights.

torch.nn supplies layers, activations and losses. **torch.optim** supplies update methods such as SGD and Adam; optimizer.step() changes parameters using their gradients. These replace separate parts of the code you wrote manually, which you can now use to check the library.

A tensor and an automatic gradient

This tiny example uses a floating-point tensor, x. The loss is (x-5)^2. The derivative is 2*(x-5), so at x=2 the gradient should be -6. Autograd calculates that derivative; it does not take a learning step for you.

```
import torch
x = torch.tensor(2.0, requires_grad=True)
loss = (x - 5.0)**2
loss.backward()
print(loss.item(), x.item(), x.grad.item())
# Expected: 9.0, 2.0, -6.0
```

Try it and explain

1. Run the example. Identify which printed number is the loss, the current value and the gradient.
2. Create a fresh tensor with x=3.0. Predict its gradient before calling backward().
3. Compare this with calculating the derivative by hand. Which part of your earlier code did backward() replace?
4. Explain why x has not changed. What additional update would move it toward 5?

The three jobs in a training loop

Forward pass: layers calculate predictions and a loss.
Backward pass: autograd calculates gradients.
Update: your own rule or an optimizer changes the parameters.

Takeaway

PyTorch is the library name; torch is how you access it in Python. Tensors hold numbers. Autograd computes gradients. An optimizer uses gradients to update parameters. You can use these parts separately, which is why your manual implementation remains useful.

9. Compare with your manual network

Small worked example

Your NumPy layer uses X @ W + b with W shaped (inputs,outputs). PyTorch Linear uses a weight array shaped (outputs,inputs), so copy W.T into its weight. Matching random seeds alone does not guarantee matching parameters. Copy the actual initial arrays and use float64 for a close numerical comparison.

Implement and experiment

1. Rebuild 2-4-1 and 2-4-4-1 using PyTorch Linear and Tanh. Leave the final layer as raw logits and use mean BCEWithLogitsLoss; do not add sigmoid before this loss.
2. Copy the same initial parameters and feed the same batch. Compare probabilities, loss, one gradient from every layer and one gradient-descent update before running full training.
3. Compare your manual GD, momentum and Adam with library equivalents using matching settings and batch order. Finally calculate library gradients but apply your OWN manual optimizer update; compare again.
4. Select the final image setting using validation only. Then create make_images(120,30) once and report final test loss, accuracy and a few predictions.

Coding hint - complete the missing work

```
# In Task 9, you may use torch and autograd.
# model = Linear(2,4) -> Tanh -> Linear(4,1)
# TODO: copy each manual W.T and flattened bias into the model
# TODO: optimizer.zero_grad(); logits = model(X_t)
# TODO: mean loss, backward(), then optimizer.step()
# Use sigmoid(logits) only for probabilities/accuracy.
# Matching settings: SGD(momentum=0.9, dampening=0,
#   nesterov=False); Adam(betas=(0.9,0.999), eps=1e-8)
# Set weight_decay=0 in both; reset state for every run.
```

Check and debug

Compare gradients after transposing the library weight gradient back. If results differ, inspect exact initialization, dtype, loss averaging, activation, bias shape and batch order before blaming randomness. A wrong gradient can still produce a decreasing loss.

Explain what you learned

Which work did autograd replace? Which work did the optimizer replace? Explain one discrepancy you found or one successful numerical match, then state what the final test result can and cannot establish.

Keep your manual implementation, each comparison table/plot and short explanations in the notebook. References: your Week 9 lecture; PyTorch documentation for torch.nn.Linear, torch.optim.SGD and torch.optim.Adam (docs.pytorch.org).