Sebastian Bordt

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Deep Learning

Exercise 3 – Optimization and Normalization

Release date: November 11, 2026

1 Pen and Paper

1.1 Bias Correction in Adam

In the lecture, we introduced the Adam optimizer. Let $\mathbf{g}^t = \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)$ denote the minibatch gradient at step t. Adam keeps two exponential moving averages: $\mathbf{m}^t$ averages the gradients (as in momentum), and $\mathbf{v}^t$ averages the element-wise squared gradients (as in RMSprop). Both start at zero, $\mathbf{m}^0 = \mathbf{v}^0 = \mathbf{0}$, and are updated as

$$\mathbf{m}^{t+1} = \beta_1 \, \mathbf{m}^t + (1 - \beta_1) \, \mathbf{g}^t,$$
$$\mathbf{v}^{t+1} = \beta_2 \, \mathbf{v}^t + (1 - \beta_2) \, \mathbf{g}^t \odot \mathbf{g}^t.$$

The parameters are then updated with the mean gradient divided (element-wise) by the root of the mean squared gradient,

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \, \frac{\hat{\mathbf{m}}^{t+1}}{\sqrt{\hat{\mathbf{v}}^{t+1}} + \epsilon}, \qquad \text{where} \qquad \hat{\mathbf{m}}^{t+1} = \frac{\mathbf{m}^{t+1}}{1 - \beta_1^{t+1}}, \quad \hat{\mathbf{v}}^{t+1} = \frac{\mathbf{v}^{t+1}}{1 - \beta_2^{t+1}}.$$

The hats denote the *bias-corrected* moving averages. Below, we use the values $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

In this exercise, we derive the correction factor $1 - \beta_2^t$ and work out what it does to the update step. The derivation of $1 - \beta_1^t$ is analogous.

a) Unroll the recursion for $\mathbf{v}^t$: compute $\mathbf{v}^1$, $\mathbf{v}^2$ and $\mathbf{v}^3$, and then write $\mathbf{v}^t$ directly as a weighted sum of the squared gradients $\mathbf{g}^0 \odot \mathbf{g}^0, \ldots, \mathbf{g}^{t-1} \odot \mathbf{g}^{t-1}$.

b) Compute the sum of the weights in this weighted sum. You can use that $\sum_{j=0}^{t-1} q^j = \frac{1-q^t}{1-q}$ for $q \neq 1$. For $\beta_2 = 0.999$, how large is the sum for $t = 1$, $t = 10$, $t = 100$ and $t = 1000$? Is $\mathbf{v}^t$ a weighted *average* of the past squared gradients?

c) Now suppose that the distribution of the squared gradient does not change over time, i.e., $\mathbb{E}[\mathbf{g}^i \odot \mathbf{g}^i] = \mathbb{E}[\mathbf{g} \odot \mathbf{g}]$ for all i. Compute $\mathbb{E}[\mathbf{v}^t]$ and $\mathbb{E}[\hat{\mathbf{v}}^t]$. What does this tell you about the correction factor $1 - \beta_2^t$?

d) To see what the correction does to the update step, consider the idealized case of a constant gradient with nonzero entries, $\mathbf{g}^i = \mathbf{g}$ for all i, and ignore ϵ. Compute the step of Adam with bias correction, $\eta \, \hat{\mathbf{m}}^t / \sqrt{\hat{\mathbf{v}}^t}$, and the step of Adam without bias correction, $\eta \, \mathbf{m}^t / \sqrt{\mathbf{v}^t}$ (root and division are element-wise). By which factor do the two steps differ? Evaluate this factor for $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $t \in \{1, 10, 100, 1000\}$.

e) Both biases vanish as $t \to \infty$. After how many updates does β_1^t, respectively β_2^t, fall below 0.01? Which of the two biases affects the update more, and in which direction?

2 Coding

2.1 Batch Norm in EDF

In exercise 2, the `Sigmoid` MLP with 6 hidden layers of 64 neurons did not train with learning rate 1.0: the validation error stayed at 89.7%, the error of a constant prediction. In this exercise, we implement batch normalization in EDF and then show that this network can be trained successfully with batch normalization. By now, you should be able to derive the backward pass of a batch normalization layer yourself. However, to save you this work, we provide you all the derivatives below.

Let $x_{b,c}$ be the input of the batch normalization node, where $b = 1, \dots, B$ indexes the examples of the minibatch and c the neurons of the layer. As in the lecture, the node computes the mean and the variance of every neuron over the minibatch,

$$\mu_c = \frac{1}{B}\sum_{b=1}^{B} x_{b,c}, \qquad \sigma_c^2 = \frac{1}{B}\sum_{b=1}^{B}(x_{b,c} - \mu_c)^2,$$

normalizes the input, and applies a learnable scale γ_c and shift β_c,

$$\hat{x}_{b,c} = \frac{x_{b,c} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}, \qquad y_{b,c} = \gamma_c\,\hat{x}_{b,c} + \beta_c,$$

with $\epsilon = 10^{-5}$ and the initial values $\gamma_c = 1$ and $\beta_c = 0$. These formulas are used during training. During evaluation, the node normalizes with running averages $\bar{\mu}_c$ and $\bar{\sigma}_c^2$ instead of μ_c and σ_c^2. The running averages start at $\bar{\mu}_c = 0$ and $\bar{\sigma}_c^2 = 1$ and are updated in every forward pass during training,

$$\bar{\mu}_c \leftarrow \rho\,\bar{\mu}_c + (1-\rho)\,\mu_c, \qquad \bar{\sigma}_c^2 \leftarrow \rho\,\bar{\sigma}_c^2 + (1-\rho)\,\sigma_c^2,$$

with $\rho = 0.9$. They are not updated by gradient descent.

In the backward pass, the node receives the gradient of the loss with respect to its output, $g_{b,c} = \partial\mathcal{L}/\partial y_{b,c}$. The gradients with respect to the parameters are

$$\frac{\partial\mathcal{L}}{\partial\gamma_c} = \sum_{b=1}^{B} g_{b,c}\,\hat{x}_{b,c}, \qquad \frac{\partial\mathcal{L}}{\partial\beta_c} = \sum_{b=1}^{B} g_{b,c}.$$

With $\hat{g}_{b,c} = \gamma_c\,g_{b,c}$, the gradient with respect to the input is

$$\frac{\partial\mathcal{L}}{\partial x_{b,c}} = \frac{1}{\sqrt{\sigma_c^2 + \epsilon}}\Big(\hat{g}_{b,c} - \frac{1}{B}\sum_{b'=1}^{B}\hat{g}_{b',c} - \hat{x}_{b,c}\,\frac{1}{B}\sum_{b'=1}^{B}\hat{g}_{b',c}\,\hat{x}_{b',c}\Big).$$

The first term is the gradient that we would get if μ_c and σ_c^2 were constants. The other two terms arise because μ_c and σ_c^2 depend on all inputs of the neuron: the second term comes from the mean and the third from the variance.

We work in `ex_03_image_classification.ipynb`. The `MLP` class in `mlp.py` inserts the batch normalization node between the affine layer and the activation function of every hidden layer, so that a hidden layer computes

$$\text{activation}(\text{BatchNorm}(\mathbf{W}\mathbf{h} + \mathbf{b})).$$

With batch normalization, we need to know whether the network is trained or evaluated (because of the running averages). We use the global variable `edf.training` for this, which `train_and_test()` sets to `True` before the training epoch and to `False` before the validation pass. Throughout this exercise, we use the network from exercise 2: `Sigmoid` activation, 6 hidden layers of 64 neurons, learning rate 1.0, batch size 64 and 10 epochs of training.

a) Train the network without batch normalization and confirm that it does not train.

b) Draw the computational graph of the MLP with batch normalization for two hidden layers. Include the parameters of the batch normalization nodes.

c) Implement the `BatchNorm` class. Note that `addgrad()` of an `edf.Parameter` sums the gradient over the minibatch, as for the weights of an `Affine` node.

Can you implement the class with vectorized NumPy operations only, that is, without any `for` loops? *Hint*: You can use `np.mean()`, `np.var()` and broadcasting across the examples of the minibatch.

d) Check your backward pass with numerical differentiation, as for the activation functions in exercise 2: `check_gradients()` compares the gradients of your `backward()` function with the symmetric difference quotient. The maximal errors should be below 10^{-3}.

e) Train the network again, now with batch normalization, and plot the training and validation errors of both runs. Does it train?

f) Now we investigate what batch normalization does inside the network. We provide you with the function `plot_activations_and_gradients()` that plots, for every hidden layer, a histogram of the activations and the norm of the gradient of the loss with respect to the activations (divided by the square root of the number of neurons). Call this function on 1000 training images for a freshly initialized network without batch normalization and for the trained network of task a). Describe what you see. Why can the network not learn?

g) Perform the visualization from f) for the network with batch normalization, again for a freshly initialized network and for the trained network of task e). What has changed?

h) In the lecture, we said that a normalization layer produces outputs with zero mean and unit variance. Is this what you see in the activation histograms of the network with batch normalization? If not, why?

2.2 Introduction to PyTorch

For this exercise, we build on what we learned in the previous exercises, but now we implement everything in PyTorch, which computes the gradients for us. Our goal is to replicate the performance of our EDF network on MNIST and then to surpass it. We work in `ex_03_pytorch.ipynb`. Install PyTorch in your Python environment first (see pytorch.org). If you are new to PyTorch, have a look at the official tutorials.

a) Implement a simple MLP in PyTorch. As in exercise 2, the network has two hidden layers of 128 neurons with ReLU activations.

b) Implement the training loop and train the MLP for 10 epochs with SGD and learning rate 0.5. We hold out $10K$ of the training images as a validation set, which we use to choose hyperparameters; the test set is only used at the very end. How does the validation error compare with the error of your EDF network?

c) Find a good learning rate for SGD and for Adam: train the MLP for the learning rates given in the notebook and plot the validation error against the learning rate. Which learning rate do you choose for each optimizer?

d) Add batch normalization and layer normalization to the MLP. Train the MLP without normalization and with each of the two, with SGD and the learning rate 2. What do you observe?

e) Add residual connections to the MLP and train a network with 20 hidden layers, with and without them. What do you observe?

f) Finally, we combine all components. Add dropout to the MLP and train an MLP with three hidden layers of 512 neurons, batch normalization and dropout (probability 0.2) with AdamW (learning rate 10^{-3}, weight decay 0.01) and a cosine learning rate schedule. You should reach a validation error of roughly 1.5%. At the very end, evaluate the model on the test set.