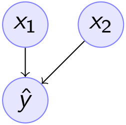
Multi-Layer Perceptrons and Deep Learning

Christos Dimitrakakis

October 21, 2025

Outline

Perceptron vs linear regression



- Network output
$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$
- Chain rule
$$\nabla_\beta L = \nabla_{\hat{y}} L \nabla_\beta \hat{y}$$
- Network gradient
$$\nabla_\beta \hat{y} = (x_1, x_2)$$

Cost functions

The only difference are the cost functions

- Perceptron
$$L = -\mathbb{I}\left\{y \neq \hat{y}\right\} \hat{y}$$

with

$$\nabla L = -\mathbb{I}\left\{y \neq \hat{y}\right\} yx$$

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

with

$$\nabla_{\hat{y}} L = 2(\hat{y} - y).$$

Layering and features

Fixed layers

- Input to layer $x \in R^n$
- Output from layer $\hat{\boldsymbol{y}} \in R^m$.

Intermediate layers

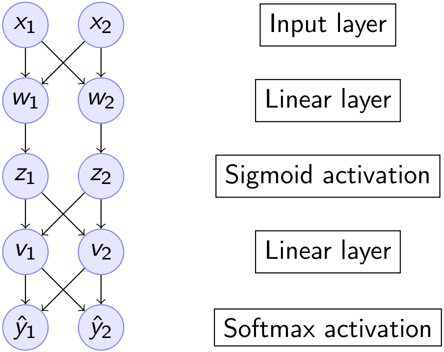
- Linear layer
- Non-linear activation function.

Linear layers types

- Dense
- Sparse
- Convolutional

Activation funnction

- Sigmoid
- Softmax



Linear layers

Example: Linear regression with n inputs, m outputs.

- Input: Features $\boldsymbol{x} \in \mathbb{R}^n$
- Dense linear layer with $\boldsymbol{B} \in \mathbb{R}^{m \times n}$
- Output: $\hat{\boldsymbol{y}} \in \mathbb{R}^m$

Dense linear layer

- Parameters $\boldsymbol{B} = \begin{pmatrix} \boldsymbol{\beta}_1 \\ \vdots \\ \boldsymbol{\beta}_m \end{pmatrix}$,
- $\boldsymbol{\beta}_i = [\theta_{i,1}, \ldots, \theta_{i,n}]$, $\boldsymbol{\beta}_i$ connects the i-th output y_i to the features $\boldsymbol{x}$:

$$y_i = \boldsymbol{\beta}_i \boldsymbol{x}$$

- In compact form:

$$\boldsymbol{y} = \boldsymbol{B}\boldsymbol{x}$$

Multiple linear layers

Repeated linear transformations are linear

It does not really help to have multiple linear layers one after the other. For example, if we transform $x \in \mathbb{R}^n$ to $z \in \mathbb{R}^k$ to $y \in \mathbb{R}^m$ through two matrices

$$z = Ax, \qquad A \in \mathbb{R}^{k \times n} \tag{1}$$
$$y = Bz, \qquad B \in \mathbb{R}^{m \times k} \tag{2}$$

We can rewrite y as

$$y = B(Ax) = (BA)x = Cx, \qquad C \in \mathbb{R}^{m \times n} \tag{3}$$

where $C = BA$.

- Successive linear layers have no advantage normally.[1]
- However, we can interlace them with non-linear activation functions.

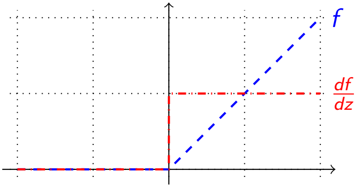
[1] Multi-task learning might be an exception.

ReLU activation

- Activation function:
$$f(x) = \max(0, x)$$
- Derivative
$$\frac{d}{dx} f(x) = \mathbb{I}\{x > 0\}$$

Typically used in the hidden layers of neural networks, as it is:

- Simple to calculate.
- Nonlinear.
- Its gradient never vanishes.



Sigmoid activation

Example: Logistic regression

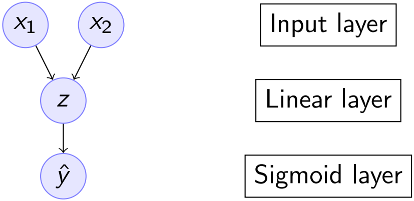
- Input $\boldsymbol{x} \in \mathbb{R}^n$
- Intermediate output: $z \in \mathbb{R}$,

$$z = \sum_{i=1}^{n} \theta_i x_i.$$

- Output: sigmoid activation $\hat{y} \in [0, 1]$.

$$f(z) = 1/[1 + \exp(-z)].$$

Now we can interpret $\hat{y} = P_{\boldsymbol{\beta}}(y = 1|x)$.



Loss function: negative log likelihood

$$\ell(\hat{y}, y) = -[\mathbb{I}\{y = 1\} \ln(\hat{y}) + \mathbb{I}\{y = -1\} \ln(1 - \hat{y})]$$

Softmax layer

Example: Multivariate logistic regression with m classes.

- Input: Features $\boldsymbol{x} \in \mathbb{R}^n$
- Fully-connected linear activation layer

$$\boldsymbol{z} = \boldsymbol{B}\boldsymbol{x}, \qquad \boldsymbol{B} \in \mathbb{R}^{m \times n}.$$

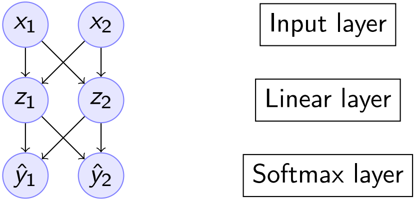
- Softmax output

$$\hat{y}_i = \frac{\exp(z_i)}{\sum_{j=1^m} \exp(z_j)}$$

We can also interpret this as

$$\hat{y}_i \triangleq \mathbb{P}(y = i \mid \boldsymbol{x})$$

with usual loss $\ell(\hat{y}, y) = -\ln \hat{y}_y$



Random projections

- Features x
- Hidden layer activation z
- Output y

Hidden layer: Random projection

Here we project the input into a high-dimensional space

$$z_i = \mathrm{sgn}(\boldsymbol{\beta}_i^\top x) = y_i$$

where $\boldsymbol{B} = [\boldsymbol{\beta}_i]_{i=1}^m$, $\theta_{i,j} \sim \mathrm{Normal}(0, 1)$

The reason for random projections

- The high dimension makes it easier to learn.
- The randomness ensures we are not learning something spurious.

Background on back-propagation

Gradient descent algorithm

- We need to minimise the expected value $\mathbb{E}_{\boldsymbol{\beta}}[L]$ of the loss function L
- Since we cannot calculate $\mathbb{E}_{\boldsymbol{\beta}}[L]$, we use:

$$\nabla_{\boldsymbol{\beta}} \mathbb{E}_{\boldsymbol{\beta}}[L] \approx \frac{1}{T} \sum_{t=1}^{T} \nabla_{\boldsymbol{\beta}} \ell(x_t, y_t, \boldsymbol{\beta}).$$

- We can then update our parameters to reduce the empirical loss

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t - \alpha_t \nabla_{\boldsymbol{\beta}} \ell(x_t, y_t, \boldsymbol{\beta}).$$

The problem

- However ℓ is a complex function of $\boldsymbol{\beta}$.
- How can we obtain $\nabla_{\boldsymbol{\beta}} \ell$?

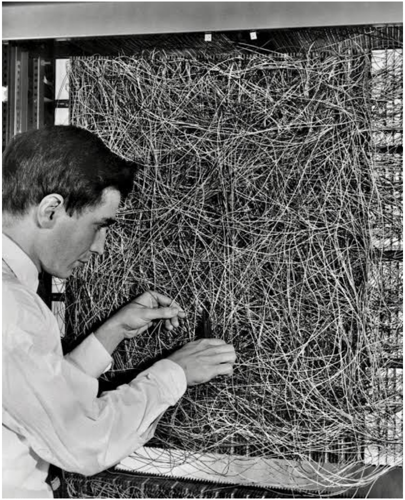
The solution

- Use the chain rule to "backpropagate" errors.

The chain rule of differentiation

[1673] Liebniz

Chain rule applied to the perceptron



[1976] Rosenblat

Chain rule for deep neural netowrks

[1982] Werbos

Backpropagation given a name

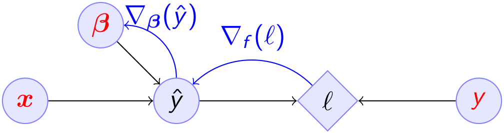
1986: Learning representations by back-propagating errors.

Rumelhart

Hinton

Williams

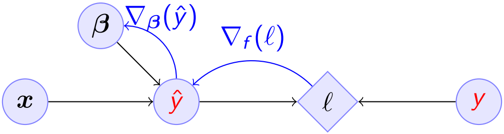
Elementary back-propagation: linear regression



- $f : X \to Y$, $\ell : Y \times Y \to \mathbb{R}$, chain rule: $\nabla_{\boldsymbol{\beta}}\ell = \nabla_{\boldsymbol{\beta}} f \nabla_{\hat{y}}\ell$
- Forward: follow the arrows to calculate variables

$$\hat{y} \triangleq f(\boldsymbol{\beta}, x) = \sum_{i=1}^{n} \theta_i x_i, \qquad \ell(\hat{y}, y) = (\hat{y} - y)^2$$

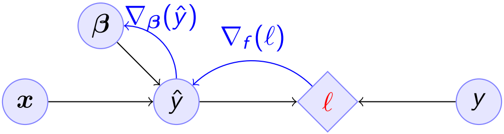
Elementary back-propagation: linear regression



- $f : X \to Y$, $\ell : Y \times Y \to \mathbb{R}$, chain rule: $\nabla_{\boldsymbol{\beta}}\ell = \nabla_{\boldsymbol{\beta}} f \nabla_{\hat{y}}\ell$
- Forward: follow the arrows to calculate variables

$$\hat{y} \triangleq f(\boldsymbol{\beta}, x) = \sum_{i=1}^{n} \theta_i x_i, \qquad \ell(\hat{y}, y) = (\hat{y} - y)^2$$

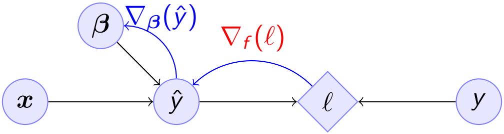
Elementary back-propagation: linear regression



- $f : X \to Y$, $\ell : Y \times Y \to \mathbb{R}$, chain rule: $\nabla_{\boldsymbol{\beta}}\ell = \nabla_{\boldsymbol{\beta}} f \nabla_{\hat{y}}\ell$
- Forward: follow the arrows to calculate variables

$$\hat{y} \triangleq f(\boldsymbol{\beta}, x) = \sum_{i=1}^{n} \theta_i x_i, \qquad \ell(\hat{y}, y) = (\hat{y} - y)^2$$

Elementary back-propagation: linear regression



- $f : X \to Y$, $\ell : Y \times Y \to \mathbb{R}$, chain rule: $\nabla_{\boldsymbol{\beta}}\ell = \nabla_{\boldsymbol{\beta}} f \nabla_{\hat{y}}\ell$
- Forward: follow the arrows to calculate variables

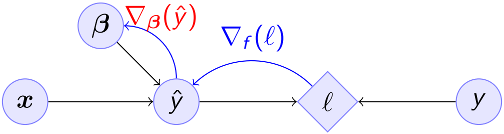
$$\hat{y} \triangleq f(\boldsymbol{\beta}, x) = \sum_{i=1}^{n} \theta_i x_i, \qquad \ell(\hat{y}, y) = (\hat{y} - y)^2$$

- Backward: return to calculate the gradients

$$\nabla_{\boldsymbol{\beta}}\ell(\hat{y}, y) = \nabla_{\boldsymbol{\beta}} f(\boldsymbol{\beta}, \boldsymbol{x}) \times \nabla_{\hat{y}}\ell(\hat{y}, y) \tag{4}$$

$$= \nabla_{\boldsymbol{\beta}} f(\boldsymbol{\beta}, \boldsymbol{x}) \times 2[\hat{y} - y] \tag{5}$$

Elementary back-propagation: linear regression



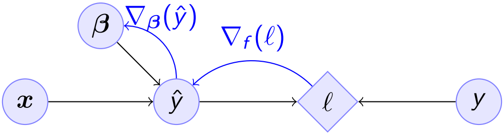
- $f : X \to Y$, $\ell : Y \times Y \to \mathbb{R}$, chain rule: $\nabla_{\boldsymbol{\beta}}\ell = \nabla_{\boldsymbol{\beta}} f \nabla_{\hat{y}}\ell$
- Forward: follow the arrows to calculate variables

$$\hat{y} \triangleq f(\boldsymbol{\beta}, x) = \sum_{i=1}^{n} \theta_i x_i, \qquad \ell(\hat{y}, y) = (\hat{y} - y)^2$$

- Backward: return to calculate the gradients

$$\begin{aligned} \nabla_{\boldsymbol{\beta}}\ell(\hat{y}, y) &= \nabla_{\boldsymbol{\beta}} f(\boldsymbol{\beta}, \boldsymbol{x}) \times \nabla_{\hat{y}}\ell(\hat{y}, y) && (4) \\ &= \nabla_{\boldsymbol{\beta}} f(\boldsymbol{\beta}, \boldsymbol{x}) \times 2[\hat{y} - y] && (5) \end{aligned}$$

Elementary back-propagation: linear regression



- $f : X \to Y$, $\ell : Y \times Y \to \mathbb{R}$, chain rule: $\nabla_{\boldsymbol{\beta}} \ell = \nabla_{\boldsymbol{\beta}} f \nabla_{\hat{y}} \ell$
- Forward: follow the arrows to calculate variables

$$\hat{y} \triangleq f(\boldsymbol{\beta}, x) = \sum_{i=1}^{n} \theta_i x_i, \qquad \ell(\hat{y}, y) = (\hat{y} - y)^2$$

- Backward: return to calculate the gradients

$$\nabla_{\boldsymbol{\beta}} \ell(\hat{y}, y) = \nabla_{\boldsymbol{\beta}} f(\boldsymbol{\beta}, \boldsymbol{x}) \times \nabla_{\hat{y}} \ell(\hat{y}, y) \tag{4}$$

$$= \nabla_{\boldsymbol{\beta}} f(\boldsymbol{\beta}, \boldsymbol{x}) \times 2[\hat{y} - y] \tag{5}$$

- Update:

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t - \alpha_t \times \nabla_{\boldsymbol{\beta}} \ell(\hat{y}_t, y_t)$$

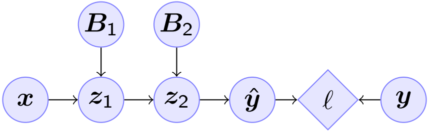
Gradient descent with *back-propagation*

- Dataset D, cost function $L = \sum_t \ell_t$
- Parameters $\boldsymbol{B}_1, \ldots, \boldsymbol{B}_k$ with k layers
- Intermediate variables: $\boldsymbol{z}_j = h_j(\boldsymbol{z}_{j-1}, \boldsymbol{B}_j)$, $\boldsymbol{z}_0 = \boldsymbol{x}$, $\boldsymbol{z}_k = \hat{\boldsymbol{y}}$.

Gradient descent with *back-propagation*

- Dataset D, cost function $L = \sum_t \ell_t$
- Parameters $\boldsymbol{B}_1, \ldots, \boldsymbol{B}_k$ with k layers
- Intermediate variables: $\boldsymbol{z}_j = h_j(\boldsymbol{z}_{j-1}, \boldsymbol{B}_j)$, $\boldsymbol{z}_0 = \boldsymbol{x}$, $\boldsymbol{z}_k = \hat{\boldsymbol{y}}$.

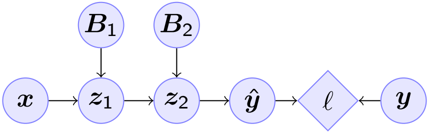
Dependency graph



Gradient descent with *back-propagation*

- Dataset D, cost function $L = \sum_t \ell_t$
- Parameters $\boldsymbol{B}_1, \ldots, \boldsymbol{B}_k$ with k layers
- Intermediate variables: $\boldsymbol{z}_j = h_j(\boldsymbol{z}_{j-1}, \boldsymbol{B}_j)$, $\boldsymbol{z}_0 = \boldsymbol{x}$, $\boldsymbol{z}_k = \hat{\boldsymbol{y}}$.

Dependency graph



Backpropagation with steepest stochastic gradient descent

- Forward step: For $j = 1, \ldots, k$, calculate $\boldsymbol{z}_j = h_j(k)$ and $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y})$
- Backward step: Calculate $\nabla_{\hat{\boldsymbol{y}}} \ell$ and $d_j \triangleq \nabla_{\boldsymbol{B}_j} \ell = \nabla_{\boldsymbol{B}_j} z_j d_{j+1}$ for $j = k \ldots, 1$
- Apply gradient: $\boldsymbol{B}_j -= \alpha d_j$.

Other algorithms and gradients

Natural gradient

Defined for probabilistic models

ADAM

Exponential moving average of gradient and square gradients

BFGS: Broyden–Fletcher–Goldfarb–Shanno algorithm

Newton-like method

Linear layer

Definition

This is a linear combination of inputs $x \in \mathbb{R}^n$ and parameter matrix

$$\boldsymbol{B} \in \mathbb{R}^{m \times n} \text{ where } \boldsymbol{B} = \begin{bmatrix} \boldsymbol{\beta}_1 \\ \vdots \\ \boldsymbol{\beta}_i \\ \vdots \\ \boldsymbol{\beta}_m \end{bmatrix} = \begin{bmatrix} \theta_{1,1} & \cdots & \theta_{1,j} & \cdots & \theta_{1,m} \\ \vdots & \ddots & \vdots & \ddots & \cdots \\ \theta_{i,1} & \cdots & \theta_{i,j} & \cdots & \theta_{i,m} \\ \vdots & \ddots & \ddots & \ddots & \cdots \\ \theta_{n,1} & \cdots & \theta_{i,j} & \cdots & \theta_{n,m} \end{bmatrix}$$

$$f(\boldsymbol{B}, \boldsymbol{x}) = \boldsymbol{B}\boldsymbol{x} \qquad f_i(\boldsymbol{B}, \boldsymbol{x}) = \boldsymbol{\beta}_i \cdot \boldsymbol{x} = \sum_{j=1}^{n} \theta_{i,j} x_j,$$

Gradient

Each partial derivative is simple:

$$\frac{\partial}{\partial \theta_{i,j}} f_k(\boldsymbol{B}, \boldsymbol{x}) = \sum_{k=1}^{n} \frac{\partial}{\partial \theta_{i,j}} \theta_{i,k} x_k = x_j$$

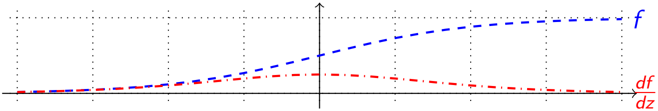
Sigmoid layer

- This layer is used for binary classification.
- It is used in the logistic regression model to obtain label probabilities.

$$f(z) = 1/(1 + \exp(-z))$$

- Derivative

$$\frac{d}{dz} f(z) = \exp(-z)/[1 + \exp(-z)]^2$$



Softmax layer

- This layer is used for multi-class classification
- It is a straightforward generalisation of the sigmoid function.

$$y_i(\boldsymbol{z}) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Derivative

$$\frac{\partial}{\partial z_i} y_i(\boldsymbol{z}) = \frac{e^{z_i} e^{\sum_{j \neq i} z_j}}{\left(\sum_j e^{z_j}\right)^2}$$

$$\frac{\partial}{\partial z_i} y_k(\boldsymbol{z}) = \frac{e^{z_i + z_k}}{\left(\sum_j e^{z_j}\right)^2}$$

Classification cost functions

Classification error

If z is the output for each class then

$$\ell(z, y) = \mathbb{I}\{y \notin \arg\max(z)\}$$

This is not differentiable.

Error margin

If z is the positive class output then

$$\ell(z, y) = -\mathbb{I}\{zy < 0\} zy$$

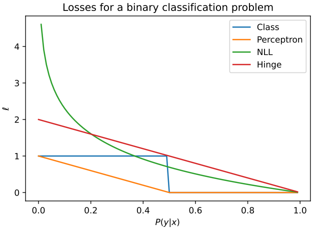
Used in the perceptron.

Negative log likelihood

If z are label probabilities, then

$$\ell(z, y) = -\ln z_y.$$

Used in logistic regression.



Hinge loss

If z are the output for each class

$$\ell(z, y) = 1 - z_y$$

Used in large margin classifiers.

Regression cost functions

L2 loss (Squared error)

If z is a prediction for y then

$$\ell(z, y) = (y - z)^2$$

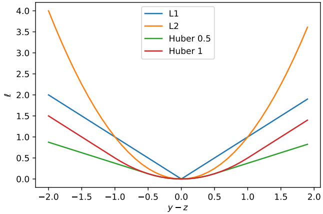
This is equivalent to negative log likelihood under Gaussianity. Used in linear regression.

L1 loss

If z is a prediction for y then

$$\ell(z, y) = |y - z|$$

Used in LASSO regression.



Huber loss

If z is a prediction, then

$$\ell(z, y) = \begin{cases} \frac{1}{2}(z - y)^2 & |z - y| \leq \delta \\ \delta(|z - y| - \frac{1}{2}\delta) & \text{otherwise.} \end{cases} \tag{6}$$

Mixes L1 and L2 losses.

Gradient descent in practice

The ideal gradient descent algorithm:

If we could calculte $\nabla_{\boldsymbol{\beta}} \mathbb{E}_{\boldsymbol{\beta}}[L]$, we could do:

$$\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n - \alpha_n \nabla_{\boldsymbol{\beta}} \mathbb{E}_{\boldsymbol{\beta}}[L]$$

for a suitable α_n schedule.

Gradient descent on the empirical error

Since we only have the data, we can try to minimse the empirical loss $\frac{1}{T}\sum_{t=1}^{T} \ell(x_t, y_t, \boldsymbol{\beta})$ through gradient descent

$$\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n - \alpha_n \frac{1}{T} \sum_{t=1}^{T} \nabla_{\boldsymbol{\beta}} \ell(x_t, y_t, \boldsymbol{\beta})$$

This is also called batch gradient descent.

Stochastic gradient descent

Gradient descent on one example:

We don't have to wait calculate $\nabla_{\boldsymbol{\beta}} \ell(x_t, y_t, \boldsymbol{\beta})$ for all t before applying the update. We can do it at every example:

$$\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n - \alpha_n \nabla_{\boldsymbol{\beta}} \ell(x_{[n]_T}, y_{[n]_T}, \boldsymbol{\beta}).$$

Here $[n]_T$ is 1 + n modulo T to ensure $n \in \{1, \ldots, T\}$.

Minibatch gradient descent

However, it is a bit better to look at K examples at a time before we change the parameters. This is called a minibatch

$$\boldsymbol{\beta}_{n+1} = \boldsymbol{\beta}_n - \alpha_n \frac{1}{K} \sum_{k=nK}^{(n+1)K-1} \nabla_{\boldsymbol{\beta}} \ell(x_{[k]_T}, y_{[k]_T}, \boldsymbol{\beta})$$

This also helps with parallelisation, since we can compute $\ell, \nabla_{\boldsymbol{\beta}} \ell$ in parallel for each example.

sklearn neural networks

Classification

Uses the cross entropy cost

```
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier(hidden_layer_sizes=(5, 2))
clf.fit(X, y)
clf.predict(X_test)
```

- Main condition is layer sizes.

Regression

```
from sklearn.neural_network import MLPRegressor
model = MLPRegressor(hidden_layer_sizes=(5, 2))
```

PyTorch

Data set-up

```
X_train = torch.tensor(X_train, dtype=torch.float32)
train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=
```

PyTorch: Manual training

Network setup

```
fc1 = nn.Linear(input_size, hidden_size)  # Input to hidden laye
fc2 = nn.Linear(hidden_size, output_size)  # Hidden layer to out
sigmoid = nn.Sigmoid() # some activation function
criterion = nn.BCELoss() #what loss to minimise
optimizer = optim.SGD(model.parameters(), lr=0.001) # how to min
```

Training

```
# Manual forward pass.
z1 = fc1(inputs)  # hidden layer 1
a1 = sigmoid(z1)      # Apply activation for hidden
z2 = fc2(a1)      # Linear combination in output layer
outputs = sigmoid(z2)  # Output layer activation
loss = criterion(outputs, labels) # Specify loss
loss.backward() # Backward pass
optimizer.step() # Update weights
```

TensorFlow

This is another library, no need to use this for this course